Ana Cavalcanti
David Deharbe
Marie-Claude Gaudel
Jim Woodcock (Eds.)

LNCS 6255

Theoretical Aspects of Computing – ICTAC 2010

7th International Colloquium
Natal, Rio Grande do Norte, Brazil, September 2010
Proceedings

Springer

Ana Cavalcanti David Deharbe
Marie-Claude Gaudel Jim Woodcock (Eds.)

Theoretical Aspects of Computing – ICTAC 2010

7th International Colloquium
Natal, Rio Grande do Norte, Brazil
September 1-3, 2010
Proceedings

Springer

Volume Editors

Ana Cavalcanti
University of York, Department of Computer Science
York YO10 5DD, United Kingdom
E-mail: ana.cavalcanti@cs.york.ac.uk

David Deharbe
Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada
Lagoa Nova 59072-970 Natal-RN, Brazil
E-mail: deharbe@gmail.com

Marie-Claude Gaudel
Université de Paris-Sud, LRI
91405, Orsay Cedex, France
E-mail: mcg@lri.fr

Jim Woodcock
University of York, Department of Computer Science
York YO10 5DD, United Kingdom
E-mail: jim@cs.york.ac.uk

Library of Congress Control Number: 2010932035

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

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

ISSN 0302-9743
ISBN-10 3-642-14807-7 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-14807-1 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

The now well-established series of International Colloquia on Theoretical Aspects of Computing (ICTAC) brings together practitioners and researchers from academia, industry and government to present research results, and exchange experience and ideas. Beyond these scholarly goals, another main purpose is to promote cooperation in research and education between participants and their institutions, from developing and industrial countries.

This volume contains the papers presented at ICTAC 2010. It was held during September 1–3 in the city of Natal, Rio Grande do Norte, Brazil.

There were 68 submissions by authors from 24 countries all around the world. Each submission was reviewed by at least three, and on average four, Program Committee members and external reviewers. After extensive discussions, they decided to accept the 23 (regular) papers presented here. Authors of a selection of these papers were invited to submit an extended version of their work to a special issue of the *Theoretical Computer Science* journal.

Seven of the papers were part of a special track including one paper on "Formal Aspects of Software Testing", and six on the "Grand Challenge in Verified Software." The special track was jointly organized by Marie-Claude Gaudel, from the Université de Paris-Sud, and Jim Woodcock, from the University of York.

The program also included invited talks. Ian Hayes, from the University of Queensland, Australia, was the FME lecturer. This volume includes his invited paper on a program algebra for sequential programs. We gratefully acknowledge the support of Formal Methods Europe in making the participation of Ian Hayes possible. A second invited paper is by Paulo Borba, from the Universidade Federal de Pernambuco, Brazil. His paper describes a refinement theory for software product lines. Stephan Merz, from INRIA, gave an invited talk on a proof assistant for TLA$^+$; the abstract for his talk is also presented here. Wolfram Schulte, from Microsoft Research, gave a talk on verification of C programs.

ICTAC 2010 was organized jointly by the Universidade Federal do Rio Grande do Norte, Brazil, and the University of York, UK. EasyChair was used to manage the submissions, their reviewing, and the proceedings production.

We are grateful to all members of the Program and Organizing Committees, and to all referees for their hard work. The support and encouragement of the Steering Committee were invaluable assets.

Finally, we would like to thank all the authors of the invited and submitted papers, and all the participants of the conference. They are the main focus of the whole event. We hope they enjoyed it.

September 2010

Ana Cavalcanti
David Déharbe

Conference Organization

Steering Committee

John Fitzgerald
Martin Leucker
Zhiming Liu (Chair)
Tobias Nipkow
Augusto Sampaio
Natarajan Shankar
Jim Woodcock

Program Chairs

Ana Cavalcanti
David Déharbe

Special Track Chairs

Marie-Claude Gaudel
Jim Woodcock

Program Committee

Bernhard Aichernig	Keijiro Araki
Jonathan Bowen	Christiano Braga
Michael Butler	Andrew Butterfield
Antonio Cerone	Jim Davies
John Fitzgerald	Wan Fokkink
Pascal Fontaine	Marcelo Frias
Lindsay Groves	Michael Hansen
Robert Hierons	Moonzoo Kim
Maciej Koutny	Pascale Le Gall
Martin Leucker	Zhiming Liu
Patrícia Machado	Ali Mili
Marius Minea	Michael Mislove
Tobias Nipkow	José Nuno Oliveira
Paritosh Pandya	Alberto Pardo
Anders Ravn	Leila Ribeiro
Markus Roggenbach	Augusto Sampaio
Bernhard Schätz	Gerhard Schellhorn
Emil Sekerinski	Natarajan Shankar

Marjan Sirjani	Jin Song Dong
Dang Van Hung	Dániel Varró
Helmut Veith	Ji Wang
Martin Wirsing	Burkhart Wolff
Husnu Yenigun	Naijun Zhan

Local Organization

David Déharbe
Anamaria Moreira
Bartira Rocha
Marcel Oliveira
Martin Musicante

External Reviewers

Nazareno Aguirre	Wilkerson Andrade
Luís Barbosa	Thomas Basuki
Andreas Bauer	Mario Benevides
Jasmin Blanchette	Harald Brandl
Simon Bumler	Emanuela Cartaxo
Pablo Castro	Zhenbang Chen
John Colley	Nam Dang
Simon Doherty	Wei Dong
Bruno Dutertre	Andrew Edmunds
Zoltán Égel	Ansgar Fehnker
Miguel Ferreira	Juan Galeotti
Alexander Gruler	Dominik Haneberg
Julian Haselmayr	Pedro Henriques
Andreas Holzer	Andras Horvath
Hans Hüttel	Visar Januzaj
Elisabeth Jöbstl	Vineet Kahlon
Narges Khakpour	Ramtin Khosravi
Yunho Kim	András Kövi
Willibald Krenn	Stefan Kugele
Shigeru Kusakabe	Christian Leuxner
Jiang Liu	Wanwei Liu
Anh Luu	Issam Maamria
Hans van Maaren	Hiroshi Mochio
Bruno Montalto	Martin Musicante
Florian Nafz	Gethin Norman
Carlos Olarte	Francisco Oliveira
Peter Ölveczky	Yoichi Omori
Sam Owre	Federica Paci
Miguel Palomino	Ngoc Pham
Pekka Pihlajasaari	Jorge Pinto

Carlos Pombo
Abdolbaghi Rezazadeh
Hassen Saïdi
Sylvain Schmitz
Martin Steffen
Volker Stolz
Daniel Thoma
Hoang Truong
Saleem Vighio
Xu Wang
Tjark Weber
Shaojie Zhang
Yu Zhang
Liang Zhao
Florian Zuleger
Viorel Preoteasa
Hamideh Sabouri
Abhisekh Sankaran
Leila Silva
Kurt Stenzel
Michael Tautschnig
Bogdan Tofan
Marcos Viera
Shuling Wang
Zhaofei Wang
James Welch
Xian Zhang
Hengjun Zhao
Manchun Zheng

Table of Contents

Invariants and Well-Foundedness in Program Algebra

Ian J. Hayes

School of Information Technology and Electrical Engineering,
The University of Queensland,
Brisbane, 4072, Australia

Abstract. Program algebras abstract the essential properties of programming languages in the form of algebraic laws. The proof of a refinement law may be expressed in terms of the algebraic properties of programs required for the law to hold, rather than directly in terms of the semantics of a language. This has the advantage that the law is then valid for any programming language that satisfies the required algebraic properties. By characterised the important properties of programming languages algebraically we can devise simple proofs of common refinement laws. In this paper we consider standard refinement laws for sequential programs. We give simple characterisations of program invariants and well foundedness of statements.

In this paper we make use of program algebras based on Kleene algebra for regular expressions [5], with extensions to incorporate tests [15] and infinite iteration [4, 17, 18]. The algebraic approach starts from a set of basic properties that are assumed—the axioms—and further properties are dervied from these. Proving laws about programs in terms of their algebraic properties has the advantage that the laws are then valid for any programming language that satisfies the axioms. The language's semantics can be used to show whether or not the axioms are valid for the language. In Section 1 we provide a brief introduce to program algebras, focussing on the commonalities between the algebras for regular expressions, relations, and programs. Section 2 details the algebraic properties of finite iteration, s^*, infinite iteration, s^∞, and finite or infinite iteration, s^ω.

Section 3 introduces program specifications based on Hoare and He's *Unifying Theories of Programming (UTP)* [13] and develops laws for reasoning about iterations and while loops in this context. Invariants play an important part in reasoning about iterations and a series of laws for invariants is developed leading to an elegant proof for the refinement law for introducing a while loop. To show termination of a loop we generalise the concept of well foundedness from relations to programs: a program is well founded if its infinite iteration has no behaviours. A simple algebraic characterisation of well foundedness leads to a simpler proof of termination in the law for a while loop. More general rules for reasoning about while loops make use of specifications that make use of relations, rather than just invariants [14]. These rules make use of iteration of a relation, r^*, as well as iteration of statements, and hence the algebraic rules for iteration come into play both for relations and statements. This leads to an elegant proof of the refinement law.

1 Regular Expressions, Binary Relations, and Programs

Regular expressions. Kleene algebra provides an algebra for regular expressions used for pattern matching strings. The left two columns of Figure 1 give the constructs of regular expressions and their meaning in terms of the language (set of strings) that they match. Concatenation of languages, $L(f) \frown L(g)$, is the set of strings formed by taking a string from $L(f)$ and another from $L(g)$ and concatenating them.

For regular expressions: $\varnothing$ is the empty (set) language, ϵ the empty string, A is the alphabet of symbols, a is a symbol (i.e., $a \in A$), and f and g are regular expressions. For a regular expression re, $L(re)$ is the corresponding language (set of strings).

Regular expression (re)	$L(re)$	Binary Relation	Program
$\varnothing$	$\{\}$	$\{\}$	$\top$
ϵ	$\{\epsilon\}$	id	1
a	$\{\langle a \rangle\}$	$\{v_1 \mapsto v_2\}$	$x := v$
			$[x = v]$
$f \mid g$	$L(f) \cup L(g)$	$r \cup w$	$s \sqcap t$
$f\,g$	$L(f) \frown L(g)$	$r \mathbin{⨟} w$	$s \,;\, t$
f^0	$\{\epsilon\}$	id	1
f^{k+1}	$L(f\,f^k)$	r^{k+1}	s^{k+1}
f^*	$\bigcup_{k \in \mathbb{N}} L(f^k)$	r^*	s^*
A^*	All finite strings over A	$\Sigma \times \Sigma$	$\bot$

Fig. 1. Example Kleene Algebras

Notational conventions. We represent statements by the letters s, t, and u; relations by the letters r and w; predicates by p and q; expressions by e and f; variables by x, y, and z; and values by v.

Binary relations. Column 3 of Figure 1 gives the relationship between the regular expressions and binary relations. A binary relation is modelled as a set of pairs of states both taken from the same state space Σ. Alternation in regular expressions corresponds to union of relations and has the empty relation as its identity. Concatenation in regular expressions corresponds to relational composition, where for σ and σ' from some state space Σ,

$$(\sigma, \sigma') \in (r \mathbin{⨟} w) \Leftrightarrow (\exists\, \sigma'' \bullet (\sigma, \sigma'') \in r \wedge (\sigma'', \sigma') \in w)\ .$$

Relational composition has as its identity the identity relation, id, which maps every element, x, in the state space Σ to itself. Iteration of a relation zero or more times, r^*, is defined by

$$r^* \mathrel{\widehat{=}} \bigcup_{k \in \mathbb{N}} r^k\ ,$$

where $r^0 \mathrel{\widehat{=}} \text{id}$ and $r^{k+1} \mathrel{\widehat{=}} r \mathbin{⨟} r^k$.

Programs. Column 4 of Figure 1 gives the relationship between the regular expressions and sequential programs. Alternation in regular expressions corresponds to nondeterministic choice ($s \sqcap t$) between programs and has as its identity the everywhere infeasible program magic, $\top$. Concatenation in regular expressions corresponds to sequential composition of programs and has as its identity the null statement, 1. As a shorthand we omit the ";" and write st for sequential composition. Iteration of a statement zero or more times is denoted s^*.

2 Iteration

Kleene algebra provides an iteration operator s^*, which iterates s zero or more times but only a *finite* number of times [5, 15]. A generalisation of this more appropriate for modelling programs is an iteration operator, s^ω, that iterates s zero or more times, including a (countably) infinite number of iterations [4]. For both these operators the number of iterations they take is (demonically) nondeterministic. A third iteration operator, s^∞, iterates s a (countably) infinite number of times [10, 8]. For conjunctive statements we have that $s^\omega = s^* \sqcap s^\infty$ (see Theorem 6 (isolation) below).

We first give fixed-point definitions of these operators, but then focus on their algebraic properties [3].

Definition 1 (iteration)

$$s^* \mathrel{\widehat{=}} \nu x \bullet 1 \sqcap sx \tag{1}$$
$$s^\omega \mathrel{\widehat{=}} \mu x \bullet 1 \sqcap sx \tag{2}$$
$$s^\infty \mathrel{\widehat{=}} \mu x \bullet sx \tag{3}$$

The iteration operators have corresponding folding/unfolding and induction laws.

Theorem 1 (Fold/unfold)

$$s^* = 1 \sqcap ss^* \tag{4}$$
$$s^\omega = 1 \sqcap ss^\omega \tag{5}$$
$$s^\infty = ss^\infty \tag{6}$$

Theorem 2 (induction)

$$x \sqsubseteq t \sqcap sx \Rightarrow x \sqsubseteq s^*t \tag{7}$$
$$t \sqcap sx \sqsubseteq x \Rightarrow s^\omega t \sqsubseteq x \tag{8}$$
$$sx \sqsubseteq x \Rightarrow s^\infty \sqsubseteq x \tag{9}$$

There are special cases of (7) and (8) if $t = 1$.

Theorem 3 (induction simple)

$$x \sqsubseteq 1 \sqcap sx \Rightarrow x \sqsubseteq s^* \tag{10}$$
$$1 \sqcap sx \sqsubseteq x \Rightarrow s^\omega \sqsubseteq x \tag{11}$$

Theorem 4 (monotonicity of iterations). *If $s \sqsubseteq t$ then*

$$s^* \sqsubseteq t^* \tag{12}$$
$$s^\omega \sqsubseteq t^\omega \tag{13}$$
$$s^\infty \sqsubseteq t^\infty \tag{14}$$

Proof. By $*$-induction (10), $s^* \sqsubseteq t^*$ provided $s^* \sqsubseteq 1 \sqcap ts^*$, which follows as:

s^*
$=$ by unfolding (4)
$1 \sqcap ss^*$
$\sqsubseteq$ as $s \sqsubseteq t$ and monotonicity
$1 \sqcap ts^*$

By ω-induction (8), $s^\omega \sqsubseteq t^\omega$ provided $1 \sqcap st^\omega \sqsubseteq t^\omega$ which follows as:

$1 \sqcap st^\omega$
$\sqsubseteq$ as $s \sqsubseteq t$ and monotonicity
$1 \sqcap tt^\omega$
$=$ by folding (5)
t^ω

By ∞-induction (9), $s^\infty \sqsubseteq t^\infty$ provided $st^\infty \sqsubseteq t^\infty$, which follows as:

st^∞
$\sqsubseteq$ as $s \sqsubseteq t$ and monotonicity
tt^∞
$=$ by folding (6)
t^∞ □

Theorem 5 (Infinite skip). *Infinite iteration of* 1 *gives the bottom element* $\bot$.

$$1^\infty = \bot$$

Proof. Because $\bot$ is the least element, it is sufficient to show $1^\infty \sqsubseteq \bot$. Using ∞-induction (9), to show $1^\infty \sqsubseteq \bot$ we need to show the obvious $1\bot \sqsubseteq \bot$. □

Definition 2 (conjunctive). *A statement, s, is conjunctive provided for all statements, t and u,*

$$s(t \sqcap u) = st \sqcap su \ .$$

Theorem 6 (isolation). *For conjunctive s*

$$s^\omega = s^* \sqcap s^\infty \ . \tag{15}$$

Proof. We use the fusion lemma, i.e., $h.(\mu f) = \mu g$ provided $h \circ f = g \circ h$ and h is strict and continuous. We chose

$$\begin{array}{lll} f = \lambda x \bullet sx & g = \lambda x \bullet 1 \sqcap sx & h = \lambda x \bullet s^* \sqcap x \\ \text{which gives} & & \\ \mu f = s^\infty & \mu g = s^\omega & h.(\mu f) = s^* \sqcap s^\infty \end{array}$$

giving $s^\omega = s^* \sqcap s^\infty$, provided $h \circ f = g \circ h$, that is,

$$\begin{array}{ll} & (\lambda x \bullet s^* \sqcap x) \circ (\lambda x \bullet sx) = (\lambda x \bullet 1 \sqcap sx) \circ (\lambda x \bullet s^* \sqcap x) \\ \equiv & (\lambda x \bullet s^* \sqcap sx) = (\lambda x \bullet 1 \sqcap s(s^* \sqcap x)) \end{array}$$

Simplifying the right side we get

$$\begin{array}{ll} & 1 \sqcap s(s^* \sqcap x) \\ = & \text{by Definition 2 (conjunctive)} \\ & 1 \sqcap ss^* \sqcap sx \\ = & \text{by folding (4)} \\ & s^* \sqcap sx \end{array}$$

We also require that h is strict and continuous, which hold by lattice properties. □

Theorem 7 (omega-infinite)

$$s^\infty = s^\omega \top$$

Proof. To show refinement from left to right we use the induction law for s^∞ (9), which requires one to show

$$\begin{array}{ll} & ss^\omega \top \sqsubseteq s^\omega \top \\ \equiv & \text{by unfolding (5)} \\ & ss^\omega \top \sqsubseteq (1 \sqcap ss^\omega) \top \\ \equiv & ss^\omega \top \sqsubseteq \top \sqcap ss^\omega \top \\ \equiv & ss^\omega \top \sqsubseteq ss^\omega \top \end{array}$$

Refinement from right to left uses the induction law for s^ω (8) which requires one to show

$$\begin{array}{ll} & \top \sqcap ss^\infty \sqsubseteq s^\infty \\ \equiv & ss^\infty \sqsubseteq s^\infty \end{array}$$

which follows from (6). □

Definition 3 (finite iteration). *A statement s iterated k times, written s^k, where k is a natural number, is defined by*

$$s^0 \mathrel{\widehat{=}} 1 \tag{16}$$
$$s^{k+1} \mathrel{\widehat{=}} ss^k \tag{17}$$

Theorem 8 (finite iteration commutes)

$$s^{k+1} = s^k s \tag{18}$$

Proof. From (17) it is sufficient to show $s^k s = s s^k$, which we prove by induction. For $k = 0$, $s^0 s = 1s = s = s1 = ss^0$. We assume for k and must show for $k+1$: $s^{k+1}s = ss^k s = sss^k = ss^{k+1}$. □

Theorem 9 (multiple unfold infinite iteration)

$$\forall k \in \mathbb{N} \bullet s^k s^\infty = s^\infty$$

Proof. We prove the property by induction. For $k = 0$, we show

$$s^0 s^\infty = 1 s^\infty = s^\infty \ .$$

We assume $s^k s^\infty = s^\infty$ holds for k, and show that $s^{k+1}s^\infty = s^\infty$.

$s^{k+1}s^\infty$
$=$ by Theorem 8 (finite iteration commutes)
$(s^k s)s^\infty$
$=$ by associativity
$s^k(ss^\infty)$
$=$ by folding (6)
$s^k s^\infty$
$=$ by induction hypothesis
s^∞ □

3 Specifications

In this section we make use of our algebraic theory for reasoning about sequential programs, with program specifications expressed as *designs* from Hoare and He's *Unifying Theory of Programming (UTP)* [13]. For programs over a state space Σ, a specification or design, $(p \vdash r)$, is given by a precondition p, which is a predicate over Σ, and a postcondition r, which is a binary relation of over $\Sigma \times \Sigma$. We assume specifications are defined as in Hoare and He's UTP theory [13]. For our purposes it is sufficient to state a number of properties of specifications in terms of assumed laws. These laws hold for UTP designs.

We write $[p]$ to state that predicate p holds for all possible values of its free variables. When writing the postcondition of a specification, we allow relations to stand for their characteristic predicates and hence allow relations to be combined using logical operators such as conjunction and disjunction, with the obvious meaning of intersection and union of the relations. Furthermore, a predicate, p, within a postcondition is interpreted as constraining the pre-state to satisfy p, and a primed predicate, p', is interpreted as constraining the post-state to satisfy p.

Law 10 (refine specification). *Provided* $[p_1 \Rightarrow p_2 \wedge (r_2 \Rightarrow r_1)]$,

$$(p_1 \vdash r_1) \sqsubseteq (p_2 \vdash r_2) \ . \tag{19}$$

Law 11 (specification sequential). *Provided* $[p \Rightarrow p_1 \wedge (r_1 \Rightarrow p_2') \wedge (r_1 \mathbin{;} r_2 \Rightarrow r)]$,

$$(p \vdash r) \sqsubseteq (p_1 \vdash r_1)(p_2 \vdash r_2)\ . \tag{20}$$

A postcondition that is the composition of two relations can be achieved by sequentially achieving each relation.

Theorem 12 (composition of relations). *Provided* $[p \wedge r_1 \Rightarrow q']$,

$$(p \vdash r_1 \mathbin{;} r_2) \sqsubseteq (p \vdash r_1)(q \vdash r_2)\ . \tag{21}$$

Proof. By Law 11 (specification sequential) we need to show

$$[p \Rightarrow p \wedge (r_1 \Rightarrow q') \wedge (r_1 \mathbin{;} r_2 \Rightarrow r_1 \mathbin{;} r_2)]$$

which follows from the assumption. □

Theorem 13 (specification disjunction)

$$(p \vdash r \vee w) \sqsubseteq (p \vdash r) \sqcap (p \vdash w) \tag{22}$$

Proof. The theorem holds provided both

$$(p \vdash r \vee w) \sqsubseteq (p \vdash r)$$
$$(p \vdash r \vee w) \sqsubseteq (p \vdash w)$$

which both hold by Law 10 (refine specification) because $r \Rightarrow r \vee w$ and $w \Rightarrow r \vee w$. □

Kozen extended Kleene algebra with tests (guards) to allow conditionals and while loops to be expressed [15]. For a predicate, b, a guard, $[b]$, acts as a null statement if b holds, and as magic ($\top$) otherwise.[1] It is a special case of a specification.

Definition 4 (guard). *For a single state predicate b,*

$$[b] \mathrel{\widehat{=}} (\mathit{true} \vdash b \wedge \mathrm{id})$$

where id *is the identity relation.*

Law 14 (separate guard)

$$(p \vdash r \wedge b') = (p \vdash r)[b] \tag{23}$$

Theorem 15 (introduce guard)

$$(p \vdash b \wedge r) \sqsubseteq [b](b \wedge p \vdash r) \tag{24}$$

[1] Note $[b]$ here is a statement, but the notation $[p]$ for a predicate p is also used to denote that p holds for all values of its free variables; this ambiguity can easily be resolved from the context it is used in.

Proof

$$\begin{array}{l} \quad (p \vdash b \wedge r) \\ \sqsubseteq \text{ by Law 11 (specification sequential)} \\ \quad (\mathit{true} \vdash \mathrm{id} \wedge b)(b \wedge p \vdash r) \\ = \text{ by Definition 4 (guard)} \\ \quad [b](b \wedge p \vdash r) \end{array}$$ □

Definition 5 (if statement). *For a predicate b and statements s_1 and s_2 an if-statement is defined as follows.*

$$\mathbf{if}\, b\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2 \mathrel{\widehat=} ([b]s_1) \sqcap ([\neg\, b]s_2)$$

Theorem 16 (introduce if)

$$(p \vdash r) \sqsubseteq \mathbf{if}\, b\, \mathbf{then} (b \wedge p \vdash r)\, \mathbf{else} (\neg\, b \wedge p \vdash r)$$

Proof

$$\begin{array}{l} \quad (p \vdash r) \\ = \text{ by Law 10 (refine specification)} \\ \quad (p \vdash (b \wedge r) \vee (\neg\, b \wedge r)) \\ \sqsubseteq \text{ by Theorem 13 (specification disjunction)} \\ \quad (p \vdash b \wedge r) \sqcap (p \vdash \neg\, b \wedge r) \\ \sqsubseteq \text{ by Theorem 15 (introduce guard) twice} \\ \quad [b](b \wedge p \vdash r) \sqcap [\neg\, b](\neg\, b \wedge p \vdash r) \\ = \text{ by Definition 5 (if statement)} \\ \quad \mathbf{if}\, b\, \mathbf{then} (b \wedge p \vdash r)\, \mathbf{else} (\neg\, b \wedge p \vdash r) \end{array}$$ □

The null or skip statement is equivalent to a specification with the identity (no change) relation, id, as its postcondition.

Law 17 (specification skip)

$$(\mathit{true} \vdash \mathrm{id}) = 1 \tag{25}$$

The null or skip statement maintains any invariant p.

Theorem 18 (skip invariant)

$$(p \vdash \mathrm{id} \wedge p') \sqsubseteq 1 \tag{26}$$

Proof

$$\begin{array}{l} \quad (p \vdash \mathrm{id} \wedge p') \\ \sqsubseteq \text{ by Law 10 (refine specification)} \\ \quad (\mathit{true} \vdash \mathrm{id}) \\ = \text{ by Law 17 (specification skip)} \\ \quad 1 \end{array}$$ □

Theorem 19 (relation with invariant)

$$(p \vdash (r_1 \mathbin{⨾} r_2) \wedge p') \sqsubseteq (p \vdash r_1 \wedge p')(p \vdash r_2 \wedge p') \tag{27}$$

Proof

$(p \vdash (r_1 \mathbin{;} r_2) \wedge p')$
$\sqsubseteq$ by Law 10 (refine specification) as $[(r_1 \wedge p') \mathbin{;} (r_2 \wedge p') \Rightarrow (r_1 \mathbin{;} r_2) \wedge p']$
$(p \vdash (r_1 \wedge p') \mathbin{;} (r_2 \wedge p'))$
$\sqsubseteq$ by Theorem 12 (composition of relations)
$(p \vdash r_1 \wedge p')(p \vdash r_2 \wedge p')$

provided $[p \wedge r_1 \wedge p' \Rightarrow p']$, which trivially holds. □

Replacing both relations r_1 and r_2 in Theorem 19 (relation with invariant) with the universal relation (*true*) which imposes no constraint, gives a rule for maintaining an invariant over a sequential composition by maintaining it for each component of the composition.

Corollary 20 (sequential invariant)

$$(p \vdash p') \sqsubseteq (p \vdash p')(p \vdash p') \tag{28}$$

For any finite number, k, of iterations, if each iteration maintains an invariant, p, then the invariant is maintained overall.

Theorem 21 (finite iteration invariant). *For any $k \in \mathbb{N}$,*

$$(p \vdash p') \sqsubseteq (p \vdash p')^k\ . \tag{29}$$

Proof. The proof is by induction on k. For $k = 0$, we use Theorem 18 (skip invariant).

$(p \vdash p') \sqsubseteq (p \vdash \mathrm{id} \wedge p') \sqsubseteq 1 = (p \vdash p')^0$

For the inductive step we assume (29) for k, and show it holds for $k+1$.

$(p \vdash p')$
$\sqsubseteq$ by Corollary 20 (sequential invariant)
$(p \vdash p')(p \vdash p')$
$\sqsubseteq$ by the inductive assumption
$(p \vdash p')(p \vdash p')^k$
$=$ by Definition 3 (finite iteration)
$(p \vdash p')^{k+1}$ □

This theorem can be generalised to the case in which the postcondition contains an iterated relation, r^k.

Theorem 22 (finite iteration relation). *For any $k \in \mathbb{N}$,*

$$(p \vdash r^k \wedge p') \sqsubseteq (p \vdash r \wedge p')^k \tag{30}$$

Proof. The proof is by induction on k. For $k = 0$, we use Theorem 18 (skip invariant).

$(p \vdash r^0 \wedge p') = (p \vdash \mathrm{id} \wedge p') \sqsubseteq 1 = (p \vdash r \wedge p')^0$

For the inductive step we assume (30) for k, and show it holds for $k+1$.

$$\begin{array}{ll} & (p \vdash r^{k+1} \wedge p') \\ = & \text{by the definition of iteration of a relation } r^{k+1} = r \mathbin{⨟} r^k \\ & (p \vdash r \mathbin{⨟} r^k \wedge p') \\ \sqsubseteq & \text{by Theorem 19 (relation with invariant)} \\ & (p \vdash r \wedge p')(p \vdash r^k \wedge p') \\ \sqsubseteq & \text{by the inductive assumption} \\ & (p \vdash r \wedge p')(p \vdash r \wedge p')^k \\ = & \text{by Definition 3 (finite iteration)} \\ & (p \vdash r \wedge p')^{k+1} \end{array}$$ □

If an invariant is maintained by each iteration, then any finite number of iterations (zero or more) will maintain the invariant.

Theorem 23 (kleene iteration invariant)

$$(p \vdash p') \sqsubseteq (p \vdash p')^*$$

Proof. The theorem follows by $*$-induction (10) provided

$$(p \vdash p') \sqsubseteq 1 \sqcap (p \vdash p')(p \vdash p')$$

To show this we expand the left side

$$\begin{array}{ll} & (p \vdash p') \\ = & \text{as demonic choice is idempotent} \\ & (p \vdash p') \sqcap (p \vdash p') \\ \sqsubseteq & \text{by Theorem 18 (skip invariant)} \\ & 1 \sqcap (p \vdash p') \\ \sqsubseteq & \text{by Corollary 20 (sequential invariant)} \\ & 1 \sqcap (p \vdash p')(p \vdash p') \end{array}$$ □

This theorem can be generalised to the case in which the postcondition contains an iterated relation, r^*, provided each iteration achieves r. Because each iteration maintains the invariant, p, it establishes the precondition for the following iteration.

Theorem 24 (kleene iteration relation)

$$(p \vdash r^* \wedge p') \sqsubseteq (p \vdash r \wedge p')^*$$

Proof. The theorem follows by $*$-induction (10) provided

$$(p \vdash r^* \wedge p') \sqsubseteq 1 \sqcap (p \vdash r \wedge p')(p \vdash r^* \wedge p')$$

To show this we expand the left side

$$\begin{array}{l}
(p \vdash r^* \wedge p') \\
= \text{as unfolding } r^* \text{ gives } r^* = \mathrm{id} \vee (r \mathbin{;} r^*) \\
\quad (p \vdash (\mathrm{id} \vee (r \mathbin{;} r^*)) \wedge p') \\
= \text{distributing} \\
\quad (p \vdash (\mathrm{id} \wedge p') \vee ((r \mathbin{;} r^*) \wedge p')) \\
\sqsubseteq \text{by Theorem 13 (specification disjunction)} \\
\quad (p \vdash \mathrm{id} \wedge p') \sqcap (p \vdash (r \mathbin{;} r^*) \wedge p') \\
\sqsubseteq \text{by Theorem 18 (skip invariant) and Theorem 19 (relation with invariant)} \\
\quad 1 \sqcap (p \vdash r \wedge p')(p \vdash r^* \wedge p') \qquad \square
\end{array}$$

While loops. A while loop $\mathbf{do}\, b \rightarrow s\, \mathbf{od}$ can be viewed as a nondeterministic choice between its possible unrollings, which include its finite unrolling zero or more times as well as its infinite unrolling. For the zero unrolling case, the guard, b, must be initially false and the while loop is equivalent to $[\neg\, b]$; for the single unrolling case, b must be true initially, but after execution of s it must be false, and in this case the while loop is equivalent to $[b]s[\neg\, b]$; and so on.

$$\begin{array}{r}
[\neg\, b] \\
\sqcap\ [b]s\ [\neg\, b] \\
\sqcap\ [b]s\ [b]s\ [\neg\, b] \\
\sqcap\ [b]s\ [b]s\ [b]s\ [\neg\, b] \\
\vdots \\
\sqcap\ [b]s\ [b]s\ [b]s\ [b]s\ \ldots
\end{array}$$

Any execution of the while loop will correspond to one of the above alternatives. Overall this corresponds to the ω-iteration of the guard and body, $([b]s)$, followed by the negation of the guard, as captured in the following definition.

Definition 6 (while loop)

$$\mathbf{do}\, b \rightarrow s\, \mathbf{od} \mathrel{\widehat{=}} ([b]s)^{\omega}[\neg\, b]$$

Well-founded relations and statements. A relation, r, is well founded provided there does not exist an infinite sequence of states $\langle \sigma_0, \sigma_1, \sigma_2, \ldots \rangle$, such that pairs of successive states are related by r, (i.e, $(\sigma_0, \sigma_1) \in r \wedge (\sigma_1, \sigma_2) \in r \wedge \ldots$). If no such infinite sequence exists, the only solution for x in the fixed point equation, $x = r \mathbin{;} x$, for the infinite iteration of r is the empty relation, i.e., $r^{\infty} = \{\}$.

In order to reason about while loops, we define what it means for a statement to be well founded in a similar manner. The infinite iteration, s^{∞}, of a well-founded statement, s, will have no possible behaviours, i.e., it will be equivalent to magic, $\top$. More generally it is useful to define what it means for a statement to be well founded when started in a state satisfying some predicate p.

Definition 7 (well-founded statement). *A statement s is well-founded on p if*

$$(p \vdash \mathit{false}) \sqsubseteq s^{\infty}\ .$$

The standard loop rule, which makes use of well foundedness to show termination, can be expressed in terms of the loop body (including the guard) being well founded on states satisfying the precondition.

Theorem 25 (terminating loop invariant). *Provided* $([b]s)$ *is well-founded on p, s is conjunctive, and*

$$(p \vdash p') \sqsubseteq [b]s$$

then $(p \vdash p' \wedge \neg b') \sqsubseteq \mathbf{do}\, b \rightarrow s\, \mathbf{od}$.

Proof

$(p \vdash p' \wedge \neg b') \sqsubseteq \mathbf{do}\, b \rightarrow s\, \mathbf{od}$
$\equiv$ Law 14 (separate guard); Definition 6 (while loop)
$(p \vdash p')[\neg b] \sqsubseteq ([b]s)^{\omega}[\neg b]$
$\Leftarrow$ by monotonicity
$(p \vdash p') \sqsubseteq ([b]s)^{\omega}$
$\Leftarrow$ by Theorem 6 (isolation); s is conjunctive
$(p \vdash p') \sqsubseteq ([b]s)^{*} \sqcap ([b]s)^{\infty}$
$\Leftarrow$ as $([b]s)$ is well-founded on p; Definition 7 (well-founded statement)
$(p \vdash p') \sqsubseteq ([b]s)^{*} \sqcap (p \vdash \mathit{false})$
$\Leftarrow$ by Law 10 (refine specification), $(p \vdash p') \sqsubseteq (p \vdash \mathit{false})$
$(p \vdash p') \sqsubseteq ([b]s)^{*}$
$\Leftarrow$ by Theorem 23 (kleene iteration invariant)
$(p \vdash p')^{*} \sqsubseteq ([b]s)^{*}$
$\Leftarrow$ by monotonicity (12)
$(p \vdash p') \sqsubseteq [b]s$ □

Note that the proviso $(p \vdash p') \sqsubseteq [b]s$ can be rewritten as $(p \wedge b \vdash p') \sqsubseteq s$, which is in the form closer to that used in the refinement calculus [16, 2]

This theorem can be generalised to the case in which the postcondition contains a relation [14].

Theorem 26 (terminating loop relation). *Provided* $([b]s)$ *is well-founded on p, s is conjunctive, and*

$$(p \vdash r \wedge p') \sqsubseteq [b]s$$

then $(p \vdash r^{*} \wedge p' \wedge \neg b') \sqsubseteq \mathbf{do}\, b \rightarrow s\, \mathbf{od}$.

Proof

$(p \vdash r^{*} \wedge p' \wedge \neg b') \sqsubseteq \mathbf{do}\, b \rightarrow s\, \mathbf{od}$
$\equiv$ by Law 14 (separate guard); Definition 6 (while loop)
$(p \vdash r^{*} \wedge p')[\neg b] \sqsubseteq ([b]s)^{\omega}[\neg b]$

$\Leftarrow$ by monotonicity
$(p \vdash r^* \wedge p') \sqsubseteq ([b]s)^\omega$
$\equiv$ by Theorem 6 (isolation); s is conjunctive
$(p \vdash r^* \wedge p') \sqsubseteq ([b]s)^* \sqcap ([b]s)^\infty$
$\Leftarrow$ as $([b]s)$ is well-founded on p; Definition 7 (well-founded statement)
$(p \vdash r^* \wedge p') \sqsubseteq ([b]s)^* \sqcap (p \vdash \mathit{false})$
$\Leftarrow$ by Law 10 (refine specification), $(p \vdash r^* \wedge p') \sqsubseteq (p \vdash \mathit{false})$
$(p \vdash r^* \wedge p') \sqsubseteq ([b]s)^*$
$\Leftarrow$ Theorem 24 (kleene iteration relation)
$(p \vdash r \wedge p')^* \sqsubseteq ([b]s)^*$
$\Leftarrow$ by monotonicity (12)
$(p \vdash r \wedge p') \sqsubseteq [b]s$ □

We note that if a relation r is well founded on p then $(p \vdash r \wedge p')^\infty = (p \vdash \mathit{false})$. Hence if $(p \vdash r \wedge p') \sqsubseteq [b]s$, by monotonicity of iterations (14), $(p \vdash r \wedge p')^\infty \sqsubseteq ([b]s)^\infty$ and hence $(p \vdash \mathit{false}) \sqsubseteq ([b]s)^\infty$, that is, $([b]s)$ is well founded.

4 Conclusions

By defining while loops in terms of iteration operators we are able to leverage the simple algebraic properties of iteration operators to devise simple proofs of refinement laws for loops. In addition, by giving an algebraic characterisation of well foundedness for programs (rather than relations) the proof of termination for the while loop rule can be handled in an elegant manner.

By phrasing the proofs of the laws in terms of the algebraic properties of the programming constructs, the laws can be used with any language whose semantics satisfies the axioms on which the theory is based. In this paper we focussed on Hoare and He's unifying theory of programming [13], but the proofs apply equally well to refinement in VDM [14], B [1], and the refinement calculus [2, 16]. The semantics of these approaches are based on relations between before and after states (or a generalisation of this to weakest precondition predicate transformers), but the axioms also apply to the richer semantics of reactive programs whose semantics is based on a relation between before and after traces of the behaviour of the program [12].

The approach taken in this paper has been applied to reasoning about programs using the *general correctness* theory [6] leading to simpler proofs of rules for reasoning about loops in that theory. We believe the approach can be extended to simplify reasoning about loops in the real-time refinement calculus [11, 9]. The rules as given do not apply directly to real-time programs because the definition of the while loop needs to be revised to allow for the time taken for guards to be evaluated. To handle this we need to introduce the concept of a guard being idle-stable and a predicate being idle-invariant [7, 8, 10], and give algebraic characterisations of these properties.

Acknowledgements. This research was supported, in part, by the EPSRC-funded *Trustworthy Ambient Systems (TrAmS)* Platform Project and Australian Research Council

(ARC) Discovery Grant DP0987452. I would like to thank Joakim von Wright for introducing me to program algebras, and Larissa Meinicke and Steve Dunne for feedback on the ideas presented in this paper.

References

1. Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Back, R.-J.R., von Wright, J.: Refinement Calculus: A Systematic Introduction. Springer, Heidelberg (1998)
3. Back, R.-J.R., von Wright, J.: Reasoning algebraically about loops. Acta Informatica 36, 295–334 (1999)
4. Cohen, E.: Separation and reduction. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 45–59. Springer, Heidelberg (2000)
5. Conway, J.H.: Regular Algebra and Finite Machines. Chapman & Hall, Boca Raton (1971)
6. Dunne, S.E., Hayes, I.J., Galloway, A.J.: Reasoning about loops in total and general correctness. In: Butterfield, A. (ed.) UTP 2008. LNCS, vol. 5713, pp. 62–81. Springer, Heidelberg (2010)
7. Hayes, I.J.: Reasoning about real-time programs using idle-invariant assertions. In: Dong, J.S., He, J., Purvis, M. (eds.) Proceedings 7th Asia-Pacific Software Engineering Conference (APSEC 2000), pp. 16–23. IEEE Computer Society, Los Alamitos (2000)
8. Hayes, I.J.: Reasoning about real-time repetitions: Terminating and nonterminating. Science of Computer Programming 43(2-3), 161–192 (2002)
9. Hayes, I.J.: A predicative semantics for real-time refinement. In: McIver, A., Morgan, C.C. (eds.) Programming Methodology, pp. 109–133. Springer, Heidelberg (2003)
10. Hayes, I.J.: Termination of real-time programs: definitely, definitely not or maybe. In: Dunne, S.E., Stoddart, W.J. (eds.) UTP 2006. LNCS, vol. 4010, pp. 141–154. Springer, Heidelberg (2006)
11. Hayes, I.J., Utting, M.: A sequential real-time refinement calculus. Acta Informatica 37(6), 385–448 (2001)
12. Hayes, I.J., Dunne, S.E., Meinicke, L.: Unifying theories of programming that distinguish nontermination and abort. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) MPC 2010. LNCS, vol. 6120, pp. 178–194. Springer, Heidelberg (2010)
13. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice Hall, Englewood Cliffs (1998)
14. Jones, C.B.: Systematic Software Development Using VDM, 2nd edn. Prentice-Hall, Englewood Cliffs (1990)
15. Kozen, D.: Kleene algebra with tests. ACM Transactions on Programming Languages and Systems 19, 427–443 (1999)
16. Morgan, C.C.: Programming from Specifications, 2nd edn. Prentice Hall, Englewood Cliffs (1994)
17. von Wright, J.: From Kleene algebra to refinement algebra. In: Möller, B., Boiten, E. (eds.) MPC 2002. LNCS, vol. 2386, pp. 233–262. Springer, Heidelberg (2002)
18. von Wright, J.: Towards a refinement algebra. Sci. of Comp. Prog. 51, 23–45 (2004)

A Theory of Software Product Line Refinement

Paulo Borba, Leopoldo Teixeira, and Rohit Gheyi

Informatics Center – Federal University of Pernambuco
Department of Computing Systems – Federal University of Campina Grande
{phmb,lmt}@cin.ufpe.br, rohit@dsc.ufcg.edu.br

Abstract. To safely derive and evolve a software product line, it is important to have a notion of product line refactoring and its underlying refinement notion, which assures behavior preservation. In this paper we present a general theory of product line refinement by extending a previous formalization with explicit interfaces between our theory and the different languages that can be used to create product line artifacts. More important, we establish product line refinement properties that justify stepwise and compositional product line development and evolution.

1 Introduction

A software product line is a set of related software products that are generated from reusable assets. Products are related in the sense that they share common functionality. Assets correspond to components, classes, property files, and other artifacts that are composed in different ways to specify or build the different products. This kind of reuse targeted at a specific set of products can bring significant productivity and time to market improvements [PBvdL05, vdLSR07].

To obtain these benefits with reduced upfront investment, previous work [Kru02, CN01, AJC+05] proposes to minimize the initial product line (domain) analysis and development process by bootstraping existing related products into a product line. In this context it is important to rely on a notion of product line refactoring [Bor09], which provides guidance and safety for deriving a product line from existing products, and also for evolving a product line by simply improving its design or by adding new products while preserving existing ones. Product line refactoring goes beyond program refactoring notions [Opd92, Fow99, BSCC04, CB05] by considering both sets of reusable assets that not necessarily correspond to valid programs, and extra artifacts, such as feature models [KCH+90, CE00], which are necessary for automatically generating products from assets.

Instead of focusing on the stronger notion of refactoring, in this paper we focus on the underlying notion of product line refinement, which also captures behavior preservation but abstracts quality improvement. This allows us to develop a formal theory of product line refinement, extending the previous formalization [Bor09] with explicit assumptions about the different languages that can be used to create product line artifacts. More important, we establish product line refinement properties that justify safe stepwise and compositional product line development and evolution. Our theory is encoded in the Prototype Verification System (PVS) [ORS92], which provides mechanized support for formal specification and verification. All properties are proved using the PVS prover.

This text is organized as follows. Section 2 introduces basic concepts and notation for feature models and other extra product line artifacts [CE00, BB09]. Several assumptions and axioms explicitly establish the interfaces between our theory and particular languages used to describe a product line. Definitions and lemmas are introduced to formalize auxiliary concepts and properties. Following that, in Sec. 3, we discuss and formalize our notion of product line refinement. We also derive basic properties that justify stepwise product line development and evolution. Next, Sec. 4 presents the product line refinement compositionality results and their proofs. We discuss related work in Sec. 5 and conclude with Sec. 6. Finally, Appendix A contains proofs omitted in the main text.

2 Product Lines Concepts

In the product line approach formalized in this paper, automatic generation of products from assets is enabled by Feature Models and Configuration Knowledge (CK) [CE00]. A feature model specifies common and variant features among products, and is used for describing and selecting products based on the features they support. A CK relates features and assets, specifying which assets implement possible feature combinations. Hence a CK can be used to actually build a product given chosen features for that product. We now explain in more detail these two kinds of artifacts and related concepts, using examples from the Mobile Media product line [FCS+08], which contains applications – such as the one illustrated in Fig. 1 – that manipulate photo, music, and video on mobile devices.

2.1 Feature Models

A feature model is essentially represented as a tree, containing features and information about how they are related. Features basically abstract groups of

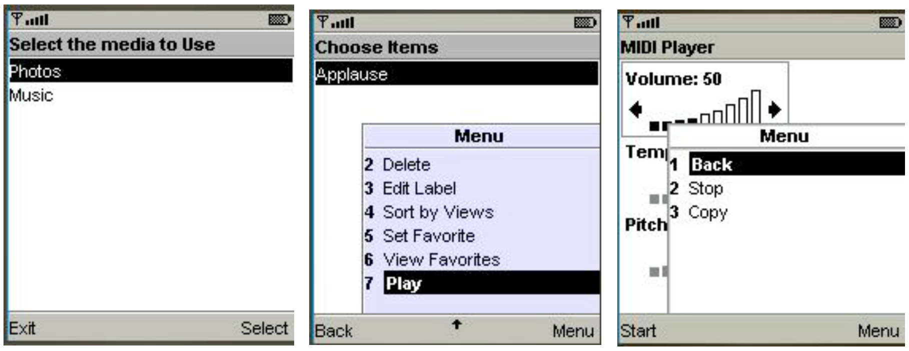


Fig. 1. Mobile Media screenshots

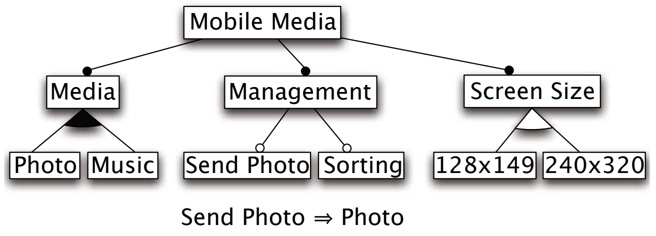


Fig. 2. Mobile Media simplified feature model

associated requirements, both functional and non-functional. In the particular feature model notation illustrated here, relationships between a parent feature and its child features (subfeatures) indicate whether the subfeatures are *optional* (present in some products but not in others, represented by an unfilled circle), *mandatory* (present in all products, represented by a filled circle), *or* (every product has at least one of them, represented by a filled triangular shape), or *alternative* (every product has exactly one of them, represented by an unfilled triangular shape). For example, Fig. 2 depicts a simplified Mobile Media feature model, where Sorting is optional, Media is mandatory, Photo and Music are or-features, and the two illustrated screen sizes are alternative.

Besides these relationships, feature models may contain propositional logic formulas about features. Feature names are used as atoms to indicate that a feature should be selected. So negation of a feature indicates that it should not be selected. For instance, the formula just below the tree in Fig. 2 states that feature Photo must be present in some product whenever feature Send Photo is selected. So

{Photo, Send Photo, 240x320},

together with the mandatory features, which hereafter we omit for brevity, is a valid feature selection (product configuration), but

{Music, Send Photo, 240x320}

is not. Likewise {Music, Photo, 240x320} is a valid configuration, but

{Music, Photo, 240x320, 128x149}

is not because it breaks the Screen Size alternative constraint. In summary, a valid configuration is one that satisfies all feature model constraints, specified both graphically and through formulas.

The set of all valid configurations often represents the semantics of a feature model. However, as different feature model notations might express constraints and configurations in different ways, our product line refinement theory abstracts the details and just assumes a generic function $[\![\_]\!]$ for obtaining the semantics of a feature model as a set of configurations.

Assumption 1. ⟨Feature model semantics⟩

$$FeatureModel : TYPE$$
$$Configuration : TYPE$$
$$[\![\_]\!] : FeatureModel \rightarrow set[Configuration]$$

We use simplified PVS notation for introducing the mentioned function and related types. In PVS, *TYPE* declares an uninterpreted type that imposes no assumptions on implementations of the specification.

As shall be clear latter, these concepts are all we require about feature models. With them, we can define our product line refinement notion and derive its properties. So our theory applies for any feature model notation whose semantics can be expressed as a set of configurations. This is the case of the feature model notation illustrated in this section and others, which have been formalized elsewhere [GMB08, AGM+06, CHE05, Bat05, SHTB07].

Given a notion of feature model semantics, it is useful to define a notion of feature model equivalence to reason about feature models. Two feature models are equivalent iff they have the same semantics.

Definition 1. ⟨Feature model equivalence⟩
Feature models F and F' are equivalent, denoted $F \cong F'$, whenever $[\![F]\!] = [\![F']\!]$.

Again, this is quite similar to the PVS specification, which defines the equivalence as a function with the following type:

$$\cong : FeatureModel, FeatureModel \rightarrow bool$$

Hereafter we omit such typing details, and overload symbols, but the types can be easily inferred from the context.

We now establish the equivalence properties for the just introduced function.

Theorem 1. ⟨Feature model equivalence – reflexivity⟩

$$\forall F : FeatureModel \cdot F \cong F$$

Proof: Follows directly from Definition 1 and the reflexivity of the equality of configuration sets. □

Theorem 2. ⟨Feature model equivalence – symmetry⟩

$$\forall F, F' : FeatureModel \cdot F \cong F' \Rightarrow F' \cong F$$

Proof: Follows directly from Definition 1 and the symmetry of the equality of configuration sets. □

Theorem 3. ⟨Feature model equivalence – transitivity⟩

$$\forall F, F', F'' : FeatureModel \cdot F \cong F' \wedge F' \cong F'' \Rightarrow F \cong F''$$

Proof: Follows directly from Definition 1 and the transitivity of the equality of configuration sets. □

These properties justify safe stepwise evolution of feature models, as illustrated in previous work [AGM+06].

2.2 Assets and Products

Besides a precise notion of feature model semantics, for defining product line refinement we assume means of comparing assets and products with respect to behavior preservation. We distinguish arbitrary asset sets ($set[Asset]$) from well-formed asset sets ($Product$), which correspond to valid products in the underlying languages used to describe assets. We assume the wf function specifies well-formedness, and $\sqsubseteq$ denotes both asset and product refinement.

Assumption 2. ⟨Asset and product refinement⟩

$$Asset : TYPE$$
$$\sqsubseteq: Asset, Asset \to bool$$
$$wf : set[Asset] \to bool$$
$$Product : TYPE = (wf)$$
$$\sqsubseteq : Product, Product \to bool$$

We use the PVS notation for defining the *Product* type as the set of all asset sets that satisfy the wf predicate.

Our product line refinement theory applies for any asset language with these notions as long as they satisfy the following properties. Both asset and product refinement must be pre-orders.

Axiom 1. ⟨Asset refinement reflexivity⟩

$$\forall a : Asset \cdot a \sqsubseteq a$$

Axiom 2. ⟨Asset refinement transitivity⟩

$$\forall a, b, c : Asset \cdot a \sqsubseteq b \wedge b \sqsubseteq c \Rightarrow a \sqsubseteq c$$

Axiom 3. ⟨Product refinement reflexivity⟩

$$\forall p : Product \cdot p \sqsubseteq p$$

Axiom 4. ⟨Product refinement transitivity⟩

$$\forall p, q, r : Product \cdot p \sqsubseteq q \wedge q \sqsubseteq r \Rightarrow p \sqsubseteq r$$

These are usually properties of any refinement notion because they are essential to support stepwise refinement and development. This is, for example, the case of existing refinement notions for object-oriented programming and modeling [BSCC04, GMB05, MGB08].

Finally, asset refinement must be compositional in the sense that refining an asset that is part of a valid product yields a refined valid product.

Axiom 5. ⟨Asset refinement compositionality⟩

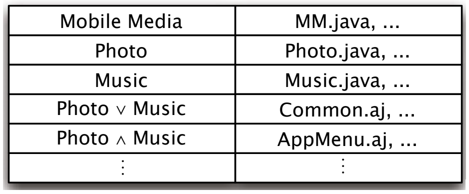
$$\begin{aligned} &\forall a, a' : Asset \cdot \forall s : set[Asset] \cdot \\ &\qquad a \sqsubseteq a' \wedge wf(a \cup s) \\ &\quad \Rightarrow wf(a' \cup s) \wedge a \cup s \sqsubseteq a' \cup s \end{aligned}$$

We use $\cup$ both to denote set union and insertion of an element to a set.

Such a compositionality property is essential to guarantee independent development of assets in a product line, and is supported, for example, by existing class refinement notions [SB04]. In that context, a product is a main command with a set of class declarations that coherently resolves all references to class and method names. In general, we do not have to limit ourselves to code assets, and consider any kind of asset that supports the concepts and properties discussed in this section.

2.3 Configuration Knowledge

As discussed in Sec. 2.1, features are groups of requirements, so they must be related to the assets that realize them. This is specified by the configuration knowledge (CK), which can be expressed in many ways, including as a relation from feature expressions (propositional formulas having feature names as atoms) to sets of asset names [BB09]. For example, showing the relation in tabular form, the following CK



Mobile Media	MM.java, ...
Photo	Photo.java, ...
Music	Music.java, ...
Photo ∨ Music	Common.aj, ...
Photo ∧ Music	AppMenu.aj, ...
⋮	⋮

establishes that if the Photo and Music features are both selected then the `AppMenu` asset, among others omitted in the fifth row, should be part of the final product. Essentially, this product line uses the `AppMenu` aspect as a variability implementation mechanism [GA01, AJC+05] that has the effect of presenting the left screenshot in Fig. 1. For usability issues, this screen should not be presented by products that have only one of the Media features, so the need for the fifth row in the simplified Mobile Media CK. Similarly, some assets are shared by the Photo and Music implementations, so we write the fourth row to avoid repeating the asset names on the second and third rows.

Given a valid product configuration, the evaluation of a CK yields the names of the assets needed to build the corresponding product. In our example, the configuration {Photo, 240x320}[1] leads to

{`MM.java`, ..., `Photo.java`, ..., `Commom.aj`, ...}.

[1] Remember we omit mandatory features for brevity.

This gives the basic intuition for the semantics of a CK. It is a function that maps product configurations into finite sets (represented by *fset*) of asset names. So our product line refinement theory relies on a CK semantic function $[\![\_]\!]$ as follows.

Assumption 3. ⟨CK semantics⟩

$$CK : TYPE$$
$$AssetName : TYPE$$
$$[\![\_]\!] : CK \rightarrow Configuration \rightarrow fset[AssetName]$$

For the CK notation illustrated in this section, the semantics of a given CK K, represented as $[\![K]\!]$, could be defined in the following way: for a configuration c, an asset name n is in the set $[\![K]\!]c$ iff there is a row in K that contains n and its expression evaluates to true according to c. But we do not give further details because our aim is to establish a product line refinement theory that is independent of CK notation, as long as this notation's semantics can be expressed as a function that maps configurations into finite sets of assets names.

Similarly to what we have done for feature models, we define a notion of CK equivalence based on the notion of CK semantics. This is useful to reason about CK. Two CK specifications are equivalent iff they have the same semantics.

Definition 2. ⟨Configuration knowledge equivalence⟩
Configuration knowledge K is equivalent to K', denoted $K \cong K'$, whenever $[\![K]\!] = [\![K']\!]$.

We now establish the equivalence properties for the just introduced relation.

Theorem 4. ⟨Configuration knowledge equivalence – reflexivity⟩

$$\forall K : CK \cdot K \cong K$$

Proof: Follows directly from Definition 2 and the reflexivity of the equality of functions. □

Theorem 5. ⟨Configuration knowledge equivalence – symmetry⟩

$$\forall K, K' : CK \cdot K \cong K' \Rightarrow K' \cong K$$

Proof: Follows directly from Definition 2 and the symmetry of the equality of functions. □

Theorem 6. ⟨Configuration knowledge equivalence – transitivity⟩

$$\forall K, K', K'' : CK \cdot K \cong K' \wedge K' \cong K'' \Rightarrow K \cong K''$$

Proof: Follows directly from Definition 2 and the transitivity of the equality of functions. □

2.4 Asset Mapping

Although the CK illustrated in the previous section refers only to code assets, in general we could also refer to requirements documents, design models, test cases, image files, XML files, and so on. For simplicity, we focus on code assets as they are equivalent to other kinds of asset for our purposes. The important issue here is not the nature of asset contents, but how the assets are compared and referred to in the CK.

We cover asset comparison in Sec. 2.2. For dealing with asset references, each product line keeps a mapping such as the following

$$\begin{array}{l}
\{\text{Main 1} \mapsto \begin{array}{l}\texttt{class Main \{}\\ \quad\texttt{...new StartUp(...);...}\\ \texttt{\}}\end{array} \\[1em]
\quad\text{Main 2} \mapsto \begin{array}{l}\texttt{class Main \{}\\ \quad\texttt{...new OnDemand(...);...}\\ \texttt{\}}\end{array} \\[1em]
\quad\text{Common.java} \mapsto \begin{array}{l}\texttt{class Common \{}\\ \quad\texttt{...}\\ \texttt{\}}\end{array} \\
\qquad\qquad\qquad\vdots \\
\}
\end{array}$$

from asset names used in a CK to actual assets. So, besides a feature model and a CK, a product line contains an asset mapping, which basically corresponds to an environment of asset declarations. This allows conflicting assets in a product line, like assets that implement alternative features, such as both `Main` classes in the illustrated asset mapping.

Formally, we specify asset mappings in PVS as follows.

Definition 3. ⟨Asset mapping⟩
Let r be a finite set of name-asset pairs ($r : fset[AssetName, Asset]$).

$$\begin{array}{l}
mapping(r) : bool = \\
\quad \forall n : AssetName \cdot \forall a, b : Asset \cdot \\
\qquad (n, a) \in r \wedge (n, b) \in r \Rightarrow a = b \\
AssetMapping : TYPE = (mapping)
\end{array}$$

Since there is not much to abstract from this notion of asset mapping, it is actually defined as part of our theory. Differently from the concepts of feature model, CK, and their semantics, the asset mapping concept is not a parameter to our theory.

We also define auxiliary functions that are used to define product line refinement. The second one is mapping application over a set. In the following, consider that $m : AssetMapping$ and $s : fset[AssetName]$.

Definition 4. ⟨Auxiliary asset mapping functions⟩

$$\begin{aligned}
&dom(m) : set[AssetName] = \\
&\quad \{n : AssetName \mid \exists a : Asset \cdot (n, a) \in m\} \\
&m\langle s\rangle : set[Asset] = \\
&\quad \{a : Asset \mid \exists n \in s \cdot (n, a) \in m\}
\end{aligned}$$

We use the notation $\exists n \in s \cdot p(n)$ as an abbreviation for the PVS notation $\exists n : AssetName \cdot n \in s \wedge p(n)$.

To derive product line refinement properties, we establish several properties of the introduced auxiliary functions. The proofs appear in Appendix A.

Lemma 1. ⟨Distributed mapping over union⟩
For asset mapping A, asset a, and finite sets of asset names S and S', if

$$a \in A\langle S \cup S'\rangle$$

then

$$a \in A\langle S\rangle \vee a \in A\langle S'\rangle \qquad \square$$

Lemma 2. ⟨Distributed mapping over singleton⟩
For asset mapping A, asset name an and finite set of asset names S, if

$$an \in dom(A)$$

then

$$\exists a : Asset \cdot (an, a) \in A \ \wedge\ A\langle an \cup S\rangle = a \cup A\langle S\rangle \qquad \square$$

Remember we use $\cup$ both for set union and insertion of an element to a set.

Lemma 3. ⟨Asset mapping domain membership⟩
For asset mapping A, asset name an and asset a, if

$$(an, a) \in A$$

then

$$an \in dom(A) \qquad \square$$

Lemma 4. ⟨Distributed mapping over set of non domain elements⟩
For asset mapping A and finite set of asset names S, if

$$\neg\exists n \in S \cdot n \in dom(A)$$

then

$$A\langle S\rangle = \{\} \qquad \square$$

For reasoning about asset mappings, we define a notion of asset mapping refinement. Asset mapping equivalence could also be defined, but we choose the weaker refinement notion since it gives us more flexibility when evolving asset mappings independently of other product line elements such as feature models and CK. As shall be clear latter, we can rely on refinement for asset mappings but not for the other elements; that is why, in previous sections, we define equivalences for them. For asset mapping refinement, exactly the same names should be mapped, not necessarily to the same assets, but to assets that refine the original ones.

Definition 5. ⟨Asset mapping refinement⟩
For asset mappings A and A', the first is refined by the second, denoted

$$A \sqsubseteq A'$$

whenever

$$\begin{array}{l} \quad dom(A) = dom(A') \\ \wedge\ \forall n \in dom(A)\cdot \\ \qquad \exists a, a' : Asset \cdot (n, a) \in A \wedge (n, a') \in A' \wedge a \sqsubseteq a' \end{array}$$

We use $\forall n \in dom(A) \cdot p(n)$ to abbreviate the PVS notation

$$\forall n : AssetName \cdot n \in dom(A) \Rightarrow p(n)$$

Note also that $a \sqsubseteq a'$ in the definition refers to asset refinement, not to program refinement.

We now prove that asset mapping refinement is a pre-order.

Theorem 7. ⟨Asset mapping refinement reflexivity⟩

$$\forall A : AssetMapping \cdot A \sqsubseteq A$$

Proof: For an arbitrary asset mapping A, from Definition 5 we have to prove that

$$\begin{array}{l} \quad dom(A) = dom(A) \\ \wedge\ \forall n \in dom(A)\cdot \\ \qquad \exists a, a' : Asset \cdot (n, a) \in A \wedge (n, a') \in A \wedge a \sqsubseteq a' \end{array}$$

The first part of the conjunction follows from equality reflexivity. For an arbitrary $n \in dom(A)$, we are left to prove

$$\exists a, a' : Asset \cdot (n, a) \in A \wedge (n, a') \in A \wedge a \sqsubseteq a' \tag{1}$$

From Definition 4, as $n \in dom(A)$, we have that

$$n \in \{n : AssetName \mid \exists a : Asset \cdot (n, a) \in A\}$$

By set comprehension and membership, we have that

$$\exists a : Asset \cdot (n, a) \in A$$

Let a_1 be such a. Then we have $(n, a_1) \in A$. From this and Axiom 1, we easily obtain 1 taking a and a' as a_1. $\square$

Theorem 8. ⟨Asset mapping refinement transitivity⟩

$$\forall A, A', A'' : \mathit{AssetMapping} \cdot A \sqsubseteq A' \wedge A' \sqsubseteq A'' \Rightarrow A \sqsubseteq A''$$

Proof: For arbitrary asset mappings A, A', and A'', assume that $A \sqsubseteq A'$ and $A' \sqsubseteq A''$. From Definition 5 we have to prove that

$$\begin{aligned} & \mathit{dom}(A) = \mathit{dom}(A'') \\ \wedge\ & \forall n \in \mathit{dom}(A) \cdot \\ & \quad \exists a, a'' : \mathit{Asset} \cdot (n, a) \in A \wedge (n, a'') \in A'' \wedge a \sqsubseteq a'' \end{aligned}$$

The first part of the conjunction follows from our assumptions, Definition 5, and equality transitivity. For an arbitrary $n \in \mathit{dom}(A)$, we are left to prove

$$\exists a, a'' : \mathit{Asset} \cdot (n, a) \in A \wedge (n, a'') \in A'' \wedge a \sqsubseteq a'' \tag{2}$$

But from our assumptions and Definition 5 we have that $n \in \mathit{dom}(A')$ and therefore

$$\begin{aligned} & (n, a) \in A \wedge (n, a') \in A' \wedge a \sqsubseteq a' \\ & (n, a') \in A' \wedge (n, a'') \in A'' \wedge a' \sqsubseteq a'' \end{aligned}$$

for some $a, a', a'' : \mathit{Asset}$. We then have the a and a'' necessary to obtain 2 directly from this and the transitivity of asset refinement (Axiom 2). $\square$

To establish the compositionality results, we rely on an important property of asset mapping refinement: if $A \sqsubseteq A'$ then products formed by using A assets are refined by products formed by corresponding A' assets.

Lemma 5. ⟨Asset mapping compositionality⟩
For asset mapping A and A', if

$$A \sqsubseteq A'$$

then

$$\begin{aligned} & \forall ans : \mathit{fset}[\mathit{AssetName}] \cdot \forall as : \mathit{fset}[\mathit{Asset}] \cdot \\ & \qquad wf(as \cup A\langle ans \rangle) \\ & \qquad \Rightarrow wf(as \cup A'\langle ans \rangle) \wedge as \cup A\langle ans \rangle \sqsubseteq as \cup A'\langle ans \rangle \end{aligned}$$ $\square$

2.5 Product Lines

We can now provide a precise definition for product lines. In particular, a product line consists of a feature model, a CK, and an asset mapping that jointly generate products, that is, valid asset sets in their target languages.

Definition 6. ⟨Product line⟩
For a feature model F, an asset mapping A, and a configuration knowledge K, we say that tuple

$$(F, A, K)$$

is a product line when, for all $c \in [\![F]\!]$,

$$wf(A\langle [\![K]\!] c \rangle)$$

We omit the PVS notation for introducing the *ProductLine* type, but it roughly corresponds to the one we use in this definition.

The well-formedness constraint in the definition is necessary because missing an entry on a CK might lead to asset sets that are missing some parts and thus are not valid products. Similarly, a mistake when writing a CK or asset mapping entry might yield an invalid asset set due to conflicting assets, like two aspects that are used as variability mechanism [GA01, AJC+05] and introduce methods with the same signature in the same class. Here we demand product line elements to be coherent as explained.

Given the importance of the well-formedness property in this definition, we establish compositionality properties related to the well-formedness function wf. First we have that feature model equivalence is compositional with respect to wf.

Lemma 6. ⟨Feature model equivalence compositionality over wf⟩
For feature models F and F', asset mapping A, and configuration knowledge K, if

$$F \cong F' \wedge \forall c \in [\![F]\!] \cdot wf(A\langle [\![K]\!] c \rangle)$$

then

$$\forall c \in [\![F']\!] \cdot wf(A\langle [\![K]\!] c \rangle)$$

□

Similarly, for CK we have the following.

Lemma 7. ⟨CK equivalence compositionality over wf⟩
For feature model F, asset mapping A, and configuration knowledge K and K', if

$$K \cong K' \wedge \forall c \in [\![F]\!] \cdot wf(A\langle [\![K]\!] c \rangle)$$

then

$$\forall c \in [\![F]\!] \cdot wf(A\langle [\![K']\!] c \rangle)$$

□

Finally, for asset mappings we have that refinement is compositional with respect to wf.

Lemma 8. ⟨Asset mapping refinement compositionality over wf⟩
For feature model F, asset mapping A and A' and configuration knowledge K, if

$$A \sqsubseteq A' \wedge \forall c \in [\![F]\!] \cdot wf(A\langle [\![K]\!]c\rangle)$$

then

$$\forall c \in [\![F]\!] \cdot wf(A'\langle [\![K]\!]c\rangle) \qquad \square$$

3 Product Line Refinement

Now that we better understand what a product line is, we can introduce a notion of product line refinement that provides guidance and safety for deriving a product line from existing products, and also for evolving a product line by simply improving its design or by adding new products while preserving existing ones.

Similar to program and model refinement [BSCC04, GMB05], product line refinement preserves behavior. However, it goes beyond source code and other kinds of reusable assets, and considers transformations to feature models and CK as well. This is illustrated by Fig. 3, where we refine the simplified Mobile Media product line by renaming the feature Music. As indicated by check marks, this renaming requires changing the feature model, CK, and asset mapping; due to a class name change, we must apply a global renaming, so the main method and other classes beyond `Music.java` are changed too.

The notion of behavior preservation should be also lifted from assets to product lines. In a product line refinement, the resulting product line should be able to generate products that behaviorally match the original product line products. So users of an original product cannot observe behavior differences when using the corresponding product of the new product line. With the renaming refinement, for example, we have only improved the product line design: the resulting

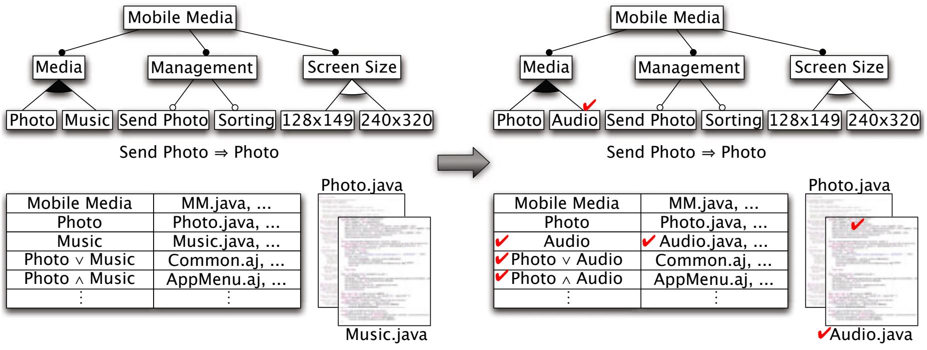
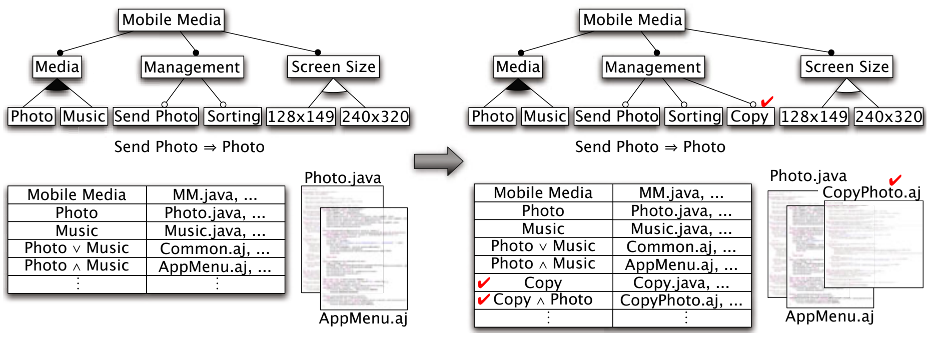


Fig. 3. Product line renaming refinement



Mobile Media	MM.java, ...
Photo	Photo.java, ...
Music	Music.java, ...
Photo ∨ Music	Common.aj, ...
Photo ∧ Music	AppMenu.aj, ...
⋮	⋮

Mobile Media	MM.java, ...
Photo	Photo.java, ...
Music	Music.java, ...
Photo ∨ Music	Common.aj, ...
Photo ∧ Music	AppMenu.aj, ...
✓ Copy	Copy.java, ...
✓ Copy ∧ Photo	CopyPhoto.aj, ...
⋮	⋮

Fig. 4. Adding an optional feature refinement

product line generates a set of products exactly equivalent to the original set. But it should not be always like that. We consider that the better product line might generate more products than the original one. As long as it generates enough products to match the original product line, users have no reason to complain. For instance, by adding the optional Copy feature (see Fig. 4), we refine our example product line. The new product line generates twice as many products as the original one, but what matters is that half of them – the ones that do not have feature Copy – behave exactly as the original products. This ensures that the transformation is safe; we extended the product line without impacting existing users.

3.1 Formalization

We formalize these ideas in terms of product refinement (see Assumption 2). Basically, each program generated by the original product line must be refined by some program of the new, improved, product line.

Definition 7. ⟨Product line refinement⟩
For product lines (F, A, K) and (F', A', K'), the first is refined by the second, denoted

$$(F, A, K) \sqsubseteq (F', A', K')$$

whenever

$$\forall c \in [\![F]\!] \cdot \exists c' \in [\![F']\!] \cdot A\langle [\![K]\!] c\rangle \sqsubseteq A'\langle [\![K']\!] c'\rangle$$

Remember that, for a configuration c, a configuration knowledge K, and an asset mapping A related to a given product line, $A\langle [\![K]\!] c\rangle$ is a well-formed set of assets. So $A\langle [\![K]\!] c\rangle \sqsubseteq A'\langle [\![K']\!] c'\rangle$ refers to the product refinement notion discussed in Sec. 2.2.

3.2 Examples and Considerations

To explore the definition just introduced, let us analyze a few concrete product line transformation scenarios.

Feature names do not matter. First let us see how the definitions applies to the transformation depicted by Fig. 3. The feature models differ only by the name of a single feature. So they generate the same set of configurations, modulo renaming. For instance, for the source (left) product line configuration {Music, 240x320} we have the target (right) product line configuration {Audio, 240x320}. As the CKs have the same structure, evaluating them with these configurations yield

$$\{\texttt{Commmon.aj}, \texttt{Music.java}, \ldots\}$$

and

$$\{\texttt{Commmon.aj}, \texttt{Audio.java}, \ldots\}.$$

The resulting sets of asset names differ at most by a single element: `Audio.java` replacing `Music.java`. Finally, when applying these sets of names to both asset mappings, we obtain the same assets modulo global renaming, which is a well known refinement for closed programs. This is precisely what, by Definition 7, we need for assuring that the source product line is refined by the target product line.

This example shows that our refinement definition focus on the product line themselves, that is, the sets of products that can be generated. Contrasting with our previous notion of feature model refactoring [AGM$^+$06], feature names do not matter. So users will not notice they are using products from the new product line, although developers might have to change their feature nomenclature when specifying product configurations. Not caring about feature names is essential for supporting useful refinements such as the just illustrated feature renaming and others that we discuss later.

Safety for existing users only. To further explore the definitions, let us consider now the transformation shown in Fig. 4. The target feature model has an extra optional feature. So it generates all configurations of the source feature model plus extensions of these configurations with feature Copy. For example, it generates both {Music, 240x320} and {Music, 240x320, Copy}. For checking refinement, we focus only on the common configurations to both feature models – configurations without Copy. As the target CK is an extension of the source CK for dealing with cases when Copy is selected, evaluating the target CK with any configuration without Copy yields the same asset names yielded by the source CK with the same configuration. In this restricted name domain, both asset mappings are equal, since the target mapping is an extension of the first for names such as `CopyPhoto.java`, which appears only when Copy is selected. Therefore, the resulting assets produced by each product line are the same, trivially implying program refinement and then product line refinement.

By focusing on the common configurations to both feature models, we check nothing about the new products offered by the new product line. In fact, they might even not operate at all. Our refinement notion assures only that users of existing products will not be disappointed by the corresponding products generated by the new product line. We give no guarantee to users of the new products, like the ones with Copy functionalities in our example. So refinements are safe transformations only in the sense that we can change a product line without impacting existing users.

Non refinements. As discussed, the transformation depicted in Fig. 3 is a refinement. Classes and aspects are transformed by a global renaming, which preserves behavior for closed programs. But suppose that, besides renaming, we change the `AppMenu.aj`[2] aspect so that, instead of the menu on the left screenshot in Fig. 1, we have a menu with "Photos" and "Audio" options. The input-output behavior of new and original products would then not match, and users would observe the difference. So we would not be able to prove program refinement, nor product line refinement, consequently.

Despite not being a refinement, this menu change is an useful product line improvement, and should be carried on. The intention, however, is to change behavior, so developers will not be able to rely on the benefits of checking refinement. The benefits of checking for refinement only apply when the intention of the transformation is to improve product line configurability or internal structure, without changing observable behavior.

3.3 Basic Properties

To support stepwise product line development and evolution, we now establish that product line refinement is a pre-order.

Theorem 9. ⟨Product line refinement reflexivity⟩

$$\forall l : \mathit{ProductLine} \cdot l \sqsubseteq l$$

Proof: Let $l = (F, A, K)$ be an arbitrary product line. By Definition 7, we have to prove that

$$\forall c \in [\![F]\!] \cdot \exists c' \in [\![F]\!] \cdot A\langle [\![K]\!] c \rangle \sqsubseteq A\langle [\![K]\!] c' \rangle$$

For an arbitrary $c \in [\![F]\!]$, just let c' be c and the proof follows from product refinement reflexivity (Axiom 3). □

Theorem 10. ⟨Product line refinement transitivity⟩

$$\forall l_1, l_2, l_3 : \mathit{ProductLine} \cdot l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3$$

[2] See Sec. 2.3 for understanding the role this aspect plays.

Proof: Let $l_1 = (F_1, A_1, K_1), l_2 = (F_2, A_2, K_2), l_3 = (F_3, A_3, K_3)$ be arbitrary product lines. Assume that $l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3$. By Definition 7, this amounts to

$$\forall c_1 \in [\![F_1]\!] \wedge \exists c_2 \in [\![F_2]\!] \cdot A_1\langle [\![K_1]\!] c_1 \rangle \sqsubseteq A_2 \langle [\![K_2]\!] c_2 \rangle \tag{3}$$

and

$$\forall c_2 \in [\![F_2]\!] \cdot \exists c_3 \in [\![F_3]\!] \cdot A_2\langle [\![K_2]\!] c_2 \rangle \sqsubseteq A_3 \langle [\![K_3]\!] c_3 \rangle \tag{4}$$

We then have to prove that

$$\forall c_1 \in [\![F_1]\!] \cdot \exists c_3 \in [\![F_3]\!] \cdot A_1\langle [\![K_1]\!] c_1 \rangle \sqsubseteq A_3 \langle [\![K_3]\!] c_3 \rangle$$

For an arbitrary $c_1 \in [\![F_1]\!]$, we have to prove that

$$\exists c_3 \in [\![F_3]\!] \cdot A_1\langle [\![K_1]\!] c_1 \rangle \sqsubseteq A_3 \langle [\![K_3]\!] c_3 \rangle \tag{5}$$

Properly instantiating c_1 in 3, we have

$$\exists c_2 \in [\![F_2]\!] \cdot A_1\langle [\![K_1]\!] c_1 \rangle \sqsubseteq A_2 \langle [\![K_2]\!] c_2 \rangle$$

Let c_2' be such c_2. Properly instantiating c_2' in 4, we have

$$\exists c_3 \in [\![F_3]\!] \cdot A_2\langle [\![K_2]\!] c_2' \rangle \sqsubseteq A_3 \langle [\![K_3]\!] c_3 \rangle$$

Let c_3' be such c_3. Then we have

$$A_1\langle [\![K_1]\!] c_1 \rangle \sqsubseteq A_2 \langle [\![K_2]\!] c_2' \rangle \wedge A_2\langle [\![K_2]\!] c_2' \rangle \sqsubseteq A_3 \langle [\![K_3]\!] c_3' \rangle$$

By product refinement transitivity (Axiom 4), we have

$$A_1\langle [\![K_1]\!] c_1 \rangle \sqsubseteq A_3 \langle [\![K_3]\!] c_3' \rangle$$

This gives us the c_3 in 5 that completes our proof. □

4 Product Line Refinement Compositionality

The product line refinement notion allows one to reason about a product line as a whole, considering its three elements (artifacts): feature model, CK, and asset mapping. However, for independent development of product line artifacts, we must support separate and compositional reasoning for each product line artifact. This allows us to evolve product line artifacts independently. We first consider feature models. Replacing a feature model by an equivalent one leads to a refined product line.

Theorem 11. ⟨Feature model equivalence compositionality⟩
For product lines (F, A, K) and (F', A, K), if

$$F \cong F'$$

then

$$(F, A, K) \sqsubseteq (F', A, K)$$

Proof: For arbitrary F, F', A, K, assume that $F \cong F'$. By Definition 7, we have to prove that

$$\forall c \in [\![F]\!] \cdot \exists c' \in [\![F']\!] \cdot A\langle [\![K]\!]c\rangle \sqsubseteq A\langle [\![K]\!]c'\rangle$$

From our assumption and Definition 1, this is equivalent to

$$\forall c \in [\![F]\!] \cdot \exists c' \in [\![F]\!] \cdot A\langle [\![K]\!]c\rangle \sqsubseteq A\langle [\![K]\!]c'\rangle$$

For an arbitrary $c \in [\![F]\!]$, just let c' be c and the proof follows from product refinement reflexivity (Axiom 4). □

We require feature model equivalence because feature model refinement, which requires $[\![F]\!] \subseteq [\![F']\!]$ instead of $[\![F]\!] = [\![F']\!]$, is not enough for ensuring that separate modifications to a feature model imply refinement for the product line. In fact, refinement allows the new feature model to have extra configurations that might not generate valid products; the associated feature model refinement transformation would not lead to a valid product line. For example, consider that the extra configurations result from eliminating an alternative constraint between two features, so that they become optional. The assets that implement these features might well be incompatible, generating an invalid program when both features are selected. Refinement of the whole product line, in this case, would also demand changes to the assets and CK.

We can also independently evolve a CK. For similar reasons, we require CK equivalence as well.

Theorem 12. ⟨CK equivalence compositionality⟩
For product lines (F, A, K) and (F, A, K'), if

$$K \cong K'$$

then

$$(F, A, K) \sqsubseteq (F, A, K')$$

Proof: The proof is similar to that of Theorem 11, using Definition 2 instead of Definition 1. □

Note that the reverse does not hold because the asset names generated by K and K' might differ for assets that have no impact on product behavior,[3] or for assets that have equivalent behavior but are named differently in the product lines. For similar reasons, the reverse does not hold for Theorem 11.

For asset mappings, we can rely only on refinement. Separately refining an asset mapping implies refinement for the product line as a whole.

[3] Obviously an anomaly, but still possible.

Theorem 13. ⟨Asset mapping refinement compositionality⟩
For product lines (F, A, K) and (F, A', K), if

$$A \sqsubseteq A'$$

then

$$(F, A, K) \sqsubseteq (F, A', K)$$

Proof: For arbitrary F, A, A', and K, assume that $A \sqsubseteq A'$. By Definition 7, we have to prove that

$$\forall c \in [\![F]\!] \cdot \exists c' \in [\![F]\!] \cdot A\langle [\![K]\!] c\rangle \sqsubseteq A'\langle [\![K]\!] c'\rangle$$

For an arbitrary $c \in [\![F]\!]$, if we prove

$$A\langle [\![K]\!] c\rangle \sqsubseteq A'\langle [\![K]\!] c\rangle \tag{6}$$

then c is the necessary c' we need to complete the proof. By Lemma 5 and our assumption, we have that

$$\begin{aligned} &\forall ans : fset[AssetName] \cdot \forall as : fset[Asset] \cdot \\ &\qquad wf(as \cup A\langle ans\rangle) \\ &\quad \Rightarrow wf(as \cup A'\langle ans\rangle) \wedge as \cup A\langle ans\rangle \sqsubseteq as \cup A'\langle ans\rangle \end{aligned} \tag{7}$$

By properly instantiating ans with $[\![K]\!] c$ and as with $\{\}$ in 7, from set union properties we obtain

$$\begin{aligned} &\qquad wf(A\langle [\![K]\!] c\rangle) \\ &\quad \Rightarrow wf(A'\langle [\![K]\!] c\rangle) \wedge A\langle [\![K]\!] c\rangle \sqsubseteq A'\langle [\![K]\!] c\rangle \end{aligned} \tag{8}$$

From Definition 6, we have that $wf(A\langle [\![K]\!] c\rangle)$ for all $c \in [\![F]\!]$. Therefore, from this and 8 we obtain

$$wf(A'\langle [\![K]\!] c\rangle) \wedge A\langle [\![K]\!] c\rangle \sqsubseteq A'\langle [\![K]\!] c\rangle$$

concluding the proof (see 6). □

Finally, we have the full compositionality theorem, which justifies completely independent development of product line artifacts.

Theorem 14. ⟨Full compositionality⟩
For product lines (F, A, K) and (F', A', K'), if

$$F \cong F' \wedge A \sqsubseteq A' \wedge K \cong K'$$

then

$$(F, A, K) \sqsubseteq (F', A', K')$$

Proof: First assume that $F \cong F'$, $A \sqsubseteq A'$, and $K \cong K'$. By Lemma 6, the fact that (F, A, K) is a product line, and Definition 6, we have that (F', A, K) is a product line. Then, using Theorem 11, we have

$$(F, A, K) \sqsubseteq (F', A, K) \tag{9}$$

Similarly, from our assumptions, deductions, and Lemma 7 we have that (F', A, K') is a product line. Using Theorem 12, we have

$$(F', A, K) \sqsubseteq (F', A, K') \tag{10}$$

Again, from our assumptions, deductions, and Lemma 8, we have that (F', A', K') is a product line. Using Theorem 13, we have

$$(F', A, K') \sqsubseteq (F', A', K') \tag{11}$$

The proof then follows from 9, 10, 11, and product line refinement transitivity (Theorem 10). □

5 Related Work

The notion of product line refinement discussed here first appeared in a product line refactoring tutorial [Bor09]. Besides talking about product line and population refactoring, this tutorial illustrates different kinds of refactoring transformation templates that can be useful for deriving and evolving product lines. In this paper we extend the initial formalization of the tutorial making clear the interface between our theory and languages used to describe product line artifacts. We also derive a number of properties that were not explored in the tutorial. We encode the theory in the PVS specification language and prove all properties with the PVS prover.

Our notion of product line refinement goes beyond refactoring of feature models [AGM$^+$06, GMB08], considering also other artifacts like configuration knowledge and assets, both in isolation and in an integrated way. In particular, the refinement notion explored here is independent of the language used to describe feature models. The cited formalization of feature models [AGM$^+$06, GMB08], and others [SHTB07], could, however, be used to instantiate our theory for dealing with specific feature model notation and semantics. Similarly, our theory is independent of product refinement notions. A program refinement notion, like the one for a sequential subset of Java [SB04, BSCC04], could be used to instantiate our general theory.

Early work [CDCvdH03] on product line refactoring focus on Product Line Architectures (PLAs) described in terms of high-level components and connectors. This work presents metrics for diagnosing structural problems in a PLA, and introduces a set of architectural refactorings that can be used to resolve these problems. Besides being specific to architectural assets, this work does not deal with other product line artifacts such as feature models and configuration knowledge. There is also no notion of behavior preservation for product lines, as captured here by our notion of product line refinement.

Several approaches [KMPY05, TBD06, LBL06, KAB07] focus on refactoring a product into a product line, not exploring product line evolution in general, as we do here. First, Kolb et al. [KMPY05] discuss a case study in refactoring legacy code components into a product line implementation. They define a systematic process for refactoring products with the aim of obtaining product lines assets. There is no discussion about feature models and configuration knowledge. Moreover, behavior preservation and configurability of the resulting product lines are only checked by testing. Similarly, Kastner et al. [KAB07] focus only on transforming code assets, implicitly relying on refinement notions for aspect-oriented programs [CB05]. As discussed here and elsewhere [Bor09] these are not adequate for justifying product line refinement and refactoring. Trujillo et al. [TBD06] go beyond code assets, but do not explicitly consider transformations to feature model and configuration knowledge. They also do not consider behavior preservation; they indeed use the term "refinement", but in the quite different sense of overriding or adding extra behavior to assets.

Liu et al. [LBL06] also focus on the process of decomposing a legacy application into features, but go further than the previously cited approaches by proposing a refactoring theory that explains how a feature can be automatically associated to a base asset (a code module, for instance) and related derivative assets, which contain feature declarations appropriate for different product configurations. Contrasting with our theory, this theory does not consider feature model transformations and assumes an implicit notion of configuration knowledge based on the idea of derivatives. So it does not consider explicit configuration knowledge transformations as we do here. Their work is, however, complementary to ours since we abstract from specific asset transformation techniques such as the one supported by their theory. By proving that their technique can be mapped to our notion of asset refinement, both theories could be used together.

The theory we present in this paper aims to formalize concepts and processes from tools [LBL06, CBS$^+$07, ACN$^+$08] and practical experience [ACV$^+$05, AJC$^+$05, KMPY05, AGM$^+$06, TBD06, KAB07] on product line refactoring. A more rigorous evaluation of the proposed theory is, however, left as future work.

6 Conclusions

In this paper we present a general theory of product line refinement, formalizing refinement and equivalence notions for product lines and its artifacts: feature model, configuration knowledge, and asset mapping. More important, we establish a number of properties that justify stepwise and compositional product line development and evolution. The presented theory is largely independent of the languages used to describe feature model, configuration knowledge, and reusable assets. We make this explicit through assumptions and axioms about basic concepts related to these languages.

By instantiating this theory with proper notations and semantic formalizations for feature models and the other product line artifacts, we can directly use the refinement and equivalence notions, and the associated properties, to guide and improve safety of the product line derivation and evolution processes. Such an instantiation also allows one to formally prove soundness of product line refactoring transformation templates [Bor09] expressed in those notations. As the transformation templates precisely specify the transformation mechanics and preconditions, their soundness is specially useful for correctly implementing the transformations and avoiding typical problems with current program refactoring tools [ST09]. In fact, soundness could help to avoid even subtler problems that can appear with product line refactoring tools.

Acknowledgements

We would like to thank colleagues of the Software Productivity Group for helping to significantly improve this work. Márcio Ribeiro provided support with the Mobile Media example. Rodrigo Bonifácio played a fundamental role developing the configuration knowledge approach used here. Vander Alves and Tiago Massoni carried on the initial ideas about feature model and product line refactoring. Fernando Castor, Carlos Pontual, Sérgio Soares, Rodrigo, and Márcio provided excellent feedback on early versions of part of the material presented here. We would also like to acknowledge current financial support from CNPq, FACEPE, and CAPES projects, and early support from FINEP and Meantime mobile creations.

A Extra Proofs

In this appendix we present the proofs we omitted in the main text. The PVS specification of the whole theory, and proof files for all lemmas and theorems are available at `http://twiki.cin.ufpe.br/twiki/bin/view/SPG/TheorySPLRefinement`.

Lemma 1. $\langle$Distributed mapping over union$\rangle$
For asset mapping A, asset a, and finite sets of asset names S and S', if

$$a \in A\langle S \cup S' \rangle$$

then

$$a \in A\langle S \rangle \vee a \in A\langle S' \rangle$$

Proof: For arbitrary A, a, S , and S', assume $a \in A\langle S \cup S' \rangle$. From this and Definition 4 ($A\langle\rangle$) we have

$$a \in \{a : Asset \mid \exists n \in S \cup S' \cdot (n, a) \in m\}$$

From set union and membership properties, we have

$$a \in \{a : Asset \mid \exists n \in S \cdot (n, a) \in m \vee \exists n \in S' \cdot (n, a) \in m\}$$

From set comprehension properties, we have

$$a \in \{a : \mathit{Asset} \mid \exists n \in S \cdot (n, a) \in m\} \cup \{a : \mathit{Asset} \mid \exists n \in S' \cdot (n, a) \in m\}$$

By applying twice Definition 4 ($A\langle\rangle$), we derive

$$a \in A\langle S\rangle \cup A\langle S'\rangle \qquad \square$$

The proof follows from the above and set membership properties.

Lemma 2. ⟨Distributed mapping over singleton⟩
For asset mapping A, asset name an, and finite set of asset names S, if

$$an \in \mathit{dom}(A)$$

then

$$\exists a : \mathit{Asset} \cdot (an, a) \in A \ \wedge \ A\langle an \cup S\rangle = a \cup A\langle S\rangle$$

Proof: For arbitrary A, an, and S, assume $an \in \mathit{dom}(A)$. From this, Definition 4 (dom), and set comprehension and membership properties, we have

$$\exists a : \mathit{Asset} \cdot (an, a) \in A \tag{12}$$

Let a_1 be such a. By Definition 4 ($A\langle\rangle$), we have

$$A\langle an \cup S\rangle = \{a : \mathit{Asset} \mid \exists n \in an \cup S \cdot (n, a) \in A\}$$

Again, by set membership and comprehension properties, we have

$$\begin{aligned} A\langle an \cup S\rangle = \\ &\{a : \mathit{Asset} \mid \exists n \in \{an\} \cdot (n, a) \in A\} \\ &\cup \{a : \mathit{Asset} \mid \exists n \in S \cdot (n, a) \in A\} \end{aligned}$$

By Definition 4 ($A\langle\rangle$), our assumption that A is an asset mapping, and set membership and comprehension properties, we have

$$A\langle \{an\} \cup S\rangle = a_1 \cup A\langle S\rangle$$

From this and remembering that 12 was instantiated with a_1, a_1 provides the a we need to conclude the proof. □

Lemma 3. ⟨Asset mapping domain membership⟩
For asset mapping A, asset name an, and asset a, if

$$(an, a) \in A$$

then

$$an \in \mathit{dom}(A)$$

Proof: For arbitrary A, an, and a, assume $(an, a) \in A$. By Definition 4 (dom), we have to prove that

$$\exists x : \mathit{Asset} \mid (an, x) \in A \qquad \square$$

Let x be a, and this concludes the proof.

Lemma 4. ⟨Distributed mapping over set of non domain elements⟩
For asset mapping A and finite set of asset names S, if

$$\neg\exists n \in S \cdot n \in \mathit{dom}(A)$$

then

$$A\langle S\rangle = \{\}$$

Proof: For arbitrary A and S, assume $\neg\exists n \in S \cdot n \in \mathit{dom}(A)$. By Definition 4 ($A\langle\rangle$), we have to prove that

$$\{a : \mathit{Asset} \mid \exists n \in S \cdot (n, a) \in A\} = \{\}$$

By Lemma 3, we then have to prove that

$$\{a : \mathit{Asset} \mid \exists n \in S \cdot n \in \mathit{dom}(A) \wedge (n, a) \in A\} = \{\}$$

The proof follows from the above, our assumption, and set comprehension properties. □

Lemma 5. ⟨Asset mapping compositionality⟩
For asset mapping A and A', if

$$A \sqsubseteq A'$$

then

$$\begin{aligned}&\forall ans : fset[AssetName] \cdot \forall as : fset[Asset] \cdot\\ &\qquad wf(as \cup A\langle ans\rangle)\\ &\quad \Rightarrow wf(as \cup A'\langle ans\rangle) \wedge as \cup A\langle ans\rangle \sqsubseteq as \cup A'\langle ans\rangle\end{aligned}$$

Proof: For arbitrary A and A', assume $A \sqsubseteq A'$. From Definition 5, we have

$$\begin{aligned}&\mathit{dom}(A) = \mathit{dom}(A')\\ &\wedge \forall n \in \mathit{dom}(A) \cdot\\ &\qquad \exists a, a' : \mathit{Asset} \cdot (n, a) \in A \wedge (n, a') \in A' \wedge a \sqsubseteq a'\end{aligned} \tag{13}$$

By induction on the cardinality of ans, assume the induction hypothesis

$$\begin{aligned}&\forall ans' : fset[AssetName] \cdot\\ &\qquad card(ans') < card(ans)\\ &\quad \Rightarrow \forall as : fset[Asset] \cdot\\ &\qquad\qquad wf(as \cup A\langle ans'\rangle)\\ &\qquad\quad \Rightarrow wf(as \cup A'\langle ans'\rangle) \wedge as \cup A\langle ans'\rangle \sqsubseteq as \cup A'\langle ans\rangle\end{aligned} \tag{14}$$

and we have to prove

$$\begin{aligned}&\forall as : fset[Asset]\cdot\\ &\qquad wf(as \cup A\langle ans\rangle)\\ &\quad \Rightarrow wf(as \cup A'\langle ans\rangle) \wedge as \cup A\langle ans\rangle \sqsubseteq as \cup A'\langle ans\rangle\end{aligned} \tag{15}$$

By case analysis, now consider that $\neg(\exists an \in ans \cdot an \in dom(A))$. By Lemma 4, we have that $A\langle ans\rangle = \{\}$. Similarly, given that $dom(A) = dom(A')$ (see 13), we also have that $A'\langle ans\rangle = \{\}$. So, by set union properties, we are left to prove that

$$\forall as : fset[Asset] \cdot wf(as) \Rightarrow wf(as) \wedge as \sqsubseteq as$$

The proof trivially follows from Axiom 3 and propositional calculus.

Let's now consider the case $\exists an \in ans \cdot an \in dom(A)$. By basic set properties, we have that $ans = an \cup ans'$ for some asset name $an \in dom(A)$ and set ans' such that $an \notin ans'$. Then, from 15, we are left to prove that

$$\begin{aligned}&\forall as : fset[Asset]\cdot\\ &\qquad wf(as \cup A\langle an \cup ans'\rangle)\\ &\quad \Rightarrow \quad wf(as \cup A'\langle an \cup ans'\rangle)\\ &\qquad \wedge\, as \cup A\langle an \cup ans'\rangle \sqsubseteq as \cup A'\langle an \cup ans'\rangle\end{aligned}$$

By Lemma 2, given that $an \in dom(A)$ and consequently $an \in dom(A')$, we have that $A\langle an \cup ans'\rangle = a \cup A\langle ans'\rangle$ and $A'\langle an \cup ans'\rangle = a' \cup A'\langle ans'\rangle$ for some assets a and a'. From 13, we also have that $a \sqsubseteq a'$. By equational reasoning, we then have to prove that

$$\begin{aligned}&\forall as : fset[Asset]\cdot\\ &\qquad wf(as \cup a \cup A\langle ans'\rangle)\\ &\quad \Rightarrow \quad wf(as \cup a' \cup A'\langle ans'\rangle)\\ &\qquad \wedge\, as \cup a \cup A\langle ans'\rangle \sqsubseteq as \cup a' \cup A'\langle ans'\rangle\end{aligned}$$

For an arbitrary as, assume $wf(as \cup a \cup A\langle ans'\rangle)$ and then we have to prove that

$$\begin{aligned}&\quad wf(as \cup a' \cup A'\langle ans'\rangle)\\ &\wedge\, as \cup a \cup A\langle ans'\rangle \sqsubseteq as \cup a' \cup A'\langle ans'\rangle\end{aligned} \tag{16}$$

By the induction hypothesis (see 14), instantiating ans' with the ans' just introduced, note that we will have $card(ans') < card(ans)$ and, therefore

$$\begin{aligned}&\forall as : fset[Asset]\cdot\\ &\qquad wf(as \cup A\langle ans'\rangle)\\ &\qquad \Rightarrow wf(as \cup A'\langle ans'\rangle) \wedge as \cup A\langle ans'\rangle \sqsubseteq as \cup A'\langle ans\rangle\end{aligned}$$

From this, instantiating as as $as \cup a$, and remembering that we have already assumed $wf(as \cup a \cup A\langle ans'\rangle)$, we have

$$\begin{aligned} & wf(as \cup A'\langle ans'\rangle) \\ \wedge\ & as \cup A\langle ans'\rangle \sqsubseteq as \cup A'\langle ans\rangle \end{aligned}$$

Now, given that $a \sqsubseteq a'$, from the compositionality axiom (Axiom 5) and the above we have that

$$\begin{aligned} & wf(()as \cup a' \cup A'\langle ans'\rangle) \\ \wedge\ & as \cup a \cup A'\langle ans'\rangle \sqsubseteq as \cup a' \cup A'\langle ans'\rangle \end{aligned}$$

The proof then follows from 16, the above, and Axiom 4. □

Lemma 6. ⟨Feature model equivalence compositionality over wf⟩
For feature models F and F', asset mapping A, and configuration knowledge K, if

$$F \cong F' \wedge \forall c \in [\![F]\!] \cdot wf(A\langle [\![K]\!] c\rangle)$$

then

$$\forall c \in [\![F']\!] \cdot wf(A\langle [\![K]\!] c\rangle)$$

Proof: For arbitrary F, F', A, and K, assume

$$F \cong F' \wedge \forall c \in [\![F]\!] \cdot wf(A\langle [\![K]\!] c\rangle)$$

By Definition 1, what we have to prove is equivalent to

$$\forall c \in [\![F]\!] \cdot wf(A\langle [\![K]\!] c\rangle)$$

which corresponds to our assumption. □

Lemma 7. ⟨CK equivalence compositionality over wf⟩
For feature model F, asset mapping A, and configuration knowledge K and K', if

$$K \cong K' \wedge \forall c \in [\![F]\!] \cdot wf(A\langle [\![K]\!] c\rangle)$$

then

$$\forall c \in [\![F]\!] \cdot wf(A\langle [\![K']\!] c\rangle)$$

Proof: Similar to proof of Lemma 6, using Definition 2 instead. □

Lemma 8. ⟨Asset mapping refinement compositionality over wf⟩
For feature model F, asset mappings A and A', and configuration knowledge K, if

$$A \sqsubseteq A' \wedge \forall c \in [\![F]\!] \cdot wf(A\langle [\![K]\!] c\rangle)$$

then

$$\forall c \in [\![F]\!] \cdot wf(A'\langle [\![K]\!] c\rangle)$$

Proof: For arbitrary F, A, A', and K, assume

$$A \sqsubseteq A' \wedge \forall c \in [\![F]\!] \cdot wf(A\langle [\![K]\!] c\rangle) \tag{17}$$

For an arbitrary $c \in [\![F]\!]$, we then have to prove that

$$wf(A'\langle [\![K]\!] c\rangle) \tag{18}$$

By properly instantiating the assumption (17) with the just introduced c, we have

$$wf(A\langle [\![K]\!] c\rangle) \tag{19}$$

From Lemma 5 and the assumption (17), we have

$$\begin{aligned}
&\forall ans : fset[AssetName] \cdot \forall as : fset[Asset] \cdot \\
&\qquad wf(as \cup A\langle ans\rangle) \\
&\quad \Rightarrow wf(as \cup A'\langle ans\rangle) \wedge \\
&\qquad as \cup A\langle ans\rangle \sqsubseteq as \cup A'\langle ans\rangle
\end{aligned}$$

Instantiating ans with $[\![K]\!] c$, as with $\{\}$, and by set union properties, we have

$$\begin{aligned}
&\qquad wf(A\langle [\![K]\!] c\rangle) \\
&\quad \Rightarrow wf(A'\langle [\![K]\!] c\rangle) \wedge A\langle [\![K]\!] c\rangle \sqsubseteq A'\langle [\![K]\!] c\rangle
\end{aligned}$$

The proof (see 18) then follows from the above and 19. □

References

[ACN+08] Alves, V., Calheiros, F., Nepomuceno, V., Menezes, A., Soares, S., Borba, P.: FLiP: Managing software product line extraction and reaction with aspects. In: 12th International Software Product Line Conference, p. 354. IEEE Computer Society, Los Alamitos (2008)

[ACV+05] Alves, V., Cardim, I., Vital, H., Sampaio, P., Damasceno, A., Borba, P., Ramalho, G.: Comparative analysis of porting strategies in J2ME games. In: 21st IEEE International Conference on Software Maintenance, pp. 123–132. IEEE Computer Society, Los Alamitos (2005)

[AGM$^+$06] Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., Lucena, C.: Refactoring product lines. In: 5th International Conference on Generative Programming and Component Engineering, pp. 201–210. ACM, New York (2006)

[AJC$^+$05] Alves, V., Matos Jr., P., Cole, L., Borba, P., Ramalho, G.: Extracting and evolving mobile games product lines. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 70–81. Springer, Heidelberg (2005)

[Bat05] Batory, D.: Feature models, grammars, and propositional formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)

[BB09] Bonifácio, R., Borba, P.: Modeling scenario variability as crosscutting mechanisms. In: 8th International Conference on Aspect-Oriented Software Development, pp. 125–136. ACM, New York (2009)

[Bor09] Borba, P.: An introduction to software product line refactoring. In: 3rd Summer School on Generative and Transformational Techniques in Software Engineering (2009) (to appear)

[BSCC04] Borba, P., Sampaio, A., Cavalcanti, A., Cornélio, M.: Algebraic reasoning for object-oriented programming. Science of Computer Programming 52, 53–100 (2004)

[CB05] Cole, L., Borba, P.: Deriving refactorings for AspectJ. In: 4th International Conference on Aspect-Oriented Software Development, pp. 123–134. ACM, New York (2005)

[CBS$^+$07] Calheiros, F., Borba, P., Soares, S., Nepomuceno, V., Alves, V.: Product line variability refactoring tool. In: 1st Workshop on Refactoring Tools, pp. 33–34 (July 2007)

[CDCvdH03] Critchlow, M., Dodd, K., Chou, J., van der Hoek, A.: Refactoring product line architectures. In: 1st International Workshop on Refactoring: Achievements, Challenges, and Effects, pp. 23–26 (2003)

[CE00] Czarnecki, K., Eisenecker, U.: Generative programming: methods, tools, and applications. Addison-Wesley, Reading (2000)

[CHE05] Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practice 10(1), 7–29 (2005)

[CN01] Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Reading (2001)

[FCS$^+$08] Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Filho, F., Dantas, F.: Evolving software product lines with aspects: an empirical study on design stability. In: 30th International Conference on Software Engineering, pp. 261–270. ACM, New York (2008)

[Fow99] Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (1999)

[GA01] Gacek, C., Anastasopoulos, M.: Implementing product line variabilities. SIGSOFT Software Engineering Notes 26(3), 109–117 (2001)

[GMB05] Gheyi, R., Massoni, T., Borba, P.: An abstract equivalence notion for object models. Electronic Notes in Theoretical Computer Science 130, 3–21 (2005)

[GMB08] Gheyi, R., Massoni, T., Borba, P.: Algebraic laws for feature models. Journal of Universal Computer Science 14(21), 3573–3591 (2008)

[KAB07] Kastner, C., Apel, S., Batory, D.: A case study implementing features using AspectJ. In: 11th International Software Product Line Conference, pp. 223–232. IEEE Computer Society, Los Alamitos (2007)

[KCH$^+$90] Kang, K., Cohen, S., Hess, J., Novak, W., Spencer Peterson, A.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990)

[KMPY05] Kolb, R., Muthig, D., Patzke, T., Yamauchi, K.: A case study in refactoring a legacy component for reuse in a product line. In: 21st International Conference on Software Maintenance, pp. 369–378. IEEE Computer Society, Los Alamitos (2005)

[Kru02] Krueger, C.: Easing the transition to software mass customization. In: van der Linden, F.J. (ed.) PFE 2002. LNCS, vol. 2290, pp. 282–293. Springer, Heidelberg (2002)

[LBL06] Liu, J., Batory, D., Lengauer, C.: Feature oriented refactoring of legacy applications. In: 28th International Conference on Software Engineering, pp. 112–121. ACM, New York (2006)

[MGB08] Massoni, T., Gheyi, R., Borba, P.: Formal model-driven program refactoring. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 362–376. Springer, Heidelberg (2008)

[Opd92] Opdyke, W.: Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois, Urbana-Champaign (1992)

[ORS92] Owre, S., Rushby, J., Shankar, N.: Pvs: A prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992)

[PBvdL05] Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Heidelberg (2005)

[SB04] Sampaio, A., Borba, P.: Transformation laws for sequential object-oriented programming. In: Cavalcanti, A., Sampaio, A., Woodcock, J. (eds.) PSSE 2004. LNCS, vol. 3167, pp. 18–63. Springer, Heidelberg (2006)

[SHTB07] Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., Bontemps, Y.: Generic semantics of feature diagrams. Computer Networks 51(2), 456–479 (2007)

[ST09] Steimann, F., Thies, A.: From public to private to absent: Refactoring Java programs under constrained accessibility. In: Drossopoulou, S. (ed.) ECOOP 2009 – Object-Oriented Programming. LNCS, vol. 5653, pp. 419–443. Springer, Heidelberg (2009)

[TBD06] Trujillo, S., Batory, D., Diaz, O.: Feature refactoring a multi-representation program into a product line. In: 5th International Conference on Generative Programming and Component Engineering, pp. 191–200. ACM, New York (2006)

[vdLSR07] van der Linden, F., Schmid, K., Rommes, E.: Software Product Lines in Action: the Best Industrial Practice in Product Line Engineering. Springer, Heidelberg (2007)

The TLA$^+$ Proof System: Building a Heterogeneous Verification Platform[⋆]

Kaustuv Chaudhuri[1], Damien Doligez[2], Leslie Lamport[3], and Stephan Merz[4]

[1] INRIA Saclay, France
[2] INRIA Rocquencourt, France
[3] Microsoft Research Silicon Valley, USA
[4] INRIA Nancy, France

Model checking has proved to be an efficient technique for finding subtle bugs in concurrent and distributed algorithms and systems. However, it is usually limited to the analysis of small instances of such systems, due to the problem of state space explosion. When model checking finds no more errors, one can attempt to verify the correctness of a model using theorem proving, which also requires efficient tool support.

TLAPS, the TLA$^+$ proof system, is a platform for the development and mechanical verification of TLA$^+$ proofs. Proofs are written in TLA$^+$, which contains a hierarchical proof language based on elementary mathematics [1]. It has been designed independently of any specific verification tool or strategy. TLAPS consists of a front-end, called the proof manager, and of a collection of back-end verifiers that include theorem provers, SMT solvers, and decision procedures. The proof manager interprets TLA$^+$ proofs and generates the corresponding proof obligations that must be verified, The current release [2] handles almost all the non-temporal part of TLA$^+$, which suffices for proving standard safety properties, but not liveness properties. The proof manager supports hierarchical and non-linear proof construction and verification so that the skeleton of an incomplete proof can be verified independently of the lower-level subproofs.

In this talk we discuss the design of the TLA$^+$ proof language and of the proof system. The different back-end verifiers used by TLAPS have complementary strengths and weaknesses, and having a heterogeneous set of proof techniques makes for a stronger overall verification system. However, it is important to ensure the overall correctness of the resulting proof. We approach this problem by making back-end verifiers proof-producing and certifying these proofs within the kernel of the interactive proof assistant Isabelle, for which we developed an encoding of the TLA$^+$ logic.

References

1. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: A TLA$^+$ proof system. In: Sutcliffe, G., Rudnicki, P., Schmidt, R., Konev, B., Schulz, S. (eds.) Proc. of the LPAR Workshop Knowledge Exchange: Automated Provers and Proof Assistants (KEAPPA 2008). CEUR Workshop Proceedings, vol. 418, pp. 17–37 (2008)
2. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: Verifying safety properties with the TLA$^+$ proof system. In: Giesl, J., Hähnle, R. (eds.) Intl. Joint Conf. Automated Reasoning (IJCAR 2010). LNCS. Springer, Heidelberg (to appear, 2010), `http://msr-inria.inria.fr/~doligez/tlaps/`

[⋆] This work is supported by the Joint Centre of Microsoft Research and INRIA in Saclay, France.

Subtyping Algorithm of Regular Tree Grammars with Disjoint Production Rules

Lei Chen and Haiming Chen

State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing 100190, China
{chl,chm}@ios.ac.cn

Abstract. Most type systems of statically typed XML processing languages are implemented based on regular expression types, where subtyping reduces to checking inclusion between tree automata, which is not efficient enough. The paper proposes the regular tree grammars with disjoint production rules and presents a subtyping method which is based on checking inclusion between regular expressions. The commonly used XML schema languages such as DTDs and XML Schemas can be described by the restricted grammars. The method works in a bottom-up way on the structures of type expressions. According to the regular expressions used in XML schema languages, different inclusion algorithms can be applied to this method. Experiments show the effectiveness of our method.

Keywords: XML, tree grammar, subtyping, algorithm.

1 Introduction

XML is a simple but flexible format for structured data that has been applied in many areas such as web services, databases, and so on. One important issue of XML processing is type checking, in which XML schema languages are regarded as types and subtype relations are checked at compile-time to ensure type correctness of programs. Many languages, such as XDuce [1,2], CDuce [3], etc., adopt unranked regular tree languages as types, which cover the commonly used schema languages. The typical way of checking subtype relation is based on tree automata, where subtyping reduces to checking inclusion between tree automata. While regular tree languages are quite powerful in expressiveness, subtyping is however expensive; the complexity is EXPTIME for the inclusion of regular tree languages or, equivalently, tree automata. There are also some other ways to do type checking [4]. However, no report shows that these type checkers are more efficient than the previous ones. Therefore, besides the work on implementation techniques to improve efficiency in practice (e.g., [5,6]), recently researches focus on finding subclasses of regular tree languages for which subtyping is more efficient.

Suzuki [7] proposes a polynomial-time algorithm for solving a subproblem of the inclusion problem for DTDs (Document Type Definitions) [8] defined by edit operations. Champavère *et al.* [9] present a polynomial-time algorithm for testing inclusion between deterministic unranked tree automata and DTDs. Complexities of decision problems for several subclasses of XML schema languages involving subclasses of regular expressions are given in [10,11]. Colazzo, Ghelli and Sartiani [12,13] give polynomial-time algorithms for a subclass of regular expressions with interleaving and numerical occurrence indicators. These investigations, of course, are far from sufficient for practice purpose. In addition, presently most of the researches focus on theoretical studies and do not use them in subtyping and give experimental results.

As opposed to the way of restricting regular expressions to obtain subclasses of regular tree languages, in this paper we define a different restriction on tree grammars, i.e., disjointness among the languages specified by any two non-terminals. As we will see in Section 2, DTDs and most actually used XML Schemas [14] fall into the restricted grammars. This restriction is orthogonal to restrictions on regular expressions with respect to subtyping. As a result, previous algorithms on various subclasses of regular expressions as mentioned above can be applied to the restricted tree grammars. We present a subtyping method for the restricted grammars. Together with using the information of labels in the restricted grammars, the method works in a bottom-up way on the structures of type expressions. A type expression is gradually reduced after checking partial subtype relations, which resembles the reduction of the production rules of grammars. The time complexity of the subtyping algorithm is in PSPACE for the restricted tree grammars with general regular expressions. It will be more efficient if regular expressions in actual schemas are simple [10] or one-unambiguous [15], the latter is recommended to use in DTDs and XML Schemas by W3C. Experiments show the effectiveness of this method.

Section 2 introduces notations and notions required in the paper. Section 3 presents the subtyping method. Section 4 describes the experiments. Section 5 gives concluding remarks.

2 Notations and Notions

The paper deals with unranked tree languages, and the term *unranked* will be omitted in the paper for simplicity when no confusion is caused.

2.1 Tree Grammars

Definition 1. *A regular tree grammar is a 4-tuple $G = (V_N, V_T, N_S, P)$, where V_N is a finite set of non-terminals, V_T is a finite set of terminals (or labels), $N_S \in V_N$ is the start symbol, P is a finite set of the production rules of the form $N \to lr$, where $N \in V_N$, $l \in V_T$, and r is a regular expression over V_N. N is the left-hand side, lr is the right-hand side, and r is the content model of this production rule.*

Given a tree grammar $G = (V_N, V_T, N_S, P)$, we write $\alpha N \beta \Rightarrow \alpha\gamma\beta$ if $N \to \gamma$ is a production rule of G and α, β are strings over the alphabet $V_N \cup V_T$. We say

α_1 *derives* α_k if $\alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_k$, or shortly, $\alpha_1 \Rightarrow^* \alpha_k$. Given a regular expression r over V_N, let $\tau(r)$ denote the set of strings, which do not contain any non-terminal, derived by r, and $L(r)$ denote the regular language specified by r.

Now we can define the regular tree grammars with disjoint production rules.

Definition 2. *A regular tree grammar with disjoint production rules (RRTG) G is a regular tree grammar in which any two production rules $N_i \rightarrow l_i r_i$ and $N_j \rightarrow l_j r_j$ satisfy either (1) $l_i \neq l_j$, or (2) $\tau(r_i) \cap \tau(r_j) = \emptyset$.*

This means that in an RRTG, the sets of strings which do not contain any non-terminal derived by any two non-terminals are disjoint. If RRTGs are applied to describe XML schema languages, any two non-terminals (as elements in schemas) define the disjoint sets of XML documents. Clearly, DTDs belong to RRTGs since the elements and labels in DTDs are in one-to-one correspondence. For XML Schemas, the type of each element is also unique in most cases since the content models of elements can be specified in two ways: either directly via complexType or simpleType, or indirectly using the type-attribute. Hence, two elements with different types define disjoint sets of XML documents. So most actually used XML Schemas can be defined by RRTGs.

Definition 2 defines the RRTGs in the normal form [16], that is, the right-hand side of each production rule must contain a label. For practical reasons, our method allows RRTGs to have some production rule of the form: $N \rightarrow r$, namely the right-hand side does not contain any label. Note that for any two production rules of the form $N_i \rightarrow r_i$ and $N_j \rightarrow r_j$, the intersection $\tau(r_i) \cap \tau(r_j)$ is not required to be empty, because the disjointness of the production rules in the normal form grammar is not affected. Moreover, each non-terminal is required to appear as the left-hand side of at most one production rule in our method. For instance, two production rules of the form $N \rightarrow r_1$ and $N \rightarrow r_2$ should be replaced with one production rule: $N \rightarrow r_1|r_2$. The resulting grammars are equivalent in expressiveness to RRTGs in the normal form. In addition, the recursive uses of non-terminals in production rules of the form $N \rightarrow r$ are allowed only occurring in the tail position of r to ensure regularity.

Example 1. Given the following tree grammar G_1 (not in the normal form), which describes a simple database of DVD store, where the lowercase words are labels and the words with an initial capital are non-terminals,

$$\begin{array}{lllll}
(1) & \mathit{Store} & \rightarrow & \mathit{store}\ (\mathit{Regulars}, \mathit{Discounts}) \\
(2) & \mathit{Regulars} & \rightarrow & \mathit{Dvd}_1 * \\
(3) & \mathit{Discounts} & \rightarrow & \mathit{Dvd}_2, \mathit{Dvd}_2 * \\
(4) & \mathit{Dvd}_1 & \rightarrow & \mathit{dvd}\ (\mathit{Title}, \mathit{Price}) \\
(5) & \mathit{Dvd}_2 & \rightarrow & \mathit{dvd}\ (\mathit{Title}, \mathit{Price}, \mathit{Dis}) \\
(5) & \mathit{Title} & \rightarrow & \mathit{title}\ \text{PCDATA} \\
(7) & \mathit{Price} & \rightarrow & \mathit{price}\ \text{PCDATA} \\
(8) & \mathit{Dis} & \rightarrow & \mathit{dis}\ \text{PCDATA}
\end{array}$$

we can have, $L(Dvd_1) = \{Dvd_1\}, \tau(Dvd_1) = \{dvd\ (title, price)\}$. As data values in XML documents, PCDATA are not considered in this paper.

Both production rules (2) and (3) of G_1 contain no label. As mentioned above, the intersection $\tau(Dvd_1*) \cap \tau(Dvd_2, Dvd_2*)$ is not required to be empty. Production rules (4) and (5) have the same label *dvd* at their right-hand sides. And $\tau(Title, Price) \cap \tau(Title, Price, Dis) = \emptyset$. So G_1 is an RRTG.

Note that not all schemas are specified by RRTGs. For instance, the following grammar G_2 is not an RRTG since the disjointness is not satisfied for the content models of the production rules (4) and (5).

$$\begin{array}{lll} (1)\ S \rightarrow s\ (A, B) & (2)\ A \rightarrow a\ C_1 & (3)\ B \rightarrow b\ C_2 \\ (4)\ C_1 \rightarrow c\ D* & (5)\ C_2 \rightarrow c\ (D, D*) & (6)\ D \rightarrow d \end{array}$$

The immediate question is that whether a given regular tree grammar is an RRTG. The case (1) in Definition 1 can be easily checked. The case (2) is a semantic restriction. Theoretically, for two production rules with the same label and content models r_1, r_2 respectively, we should check whether $\tau(r_1) \cap \tau(r_2)$ is empty. In practice, most types in XML Schemas only depend on parent context [17]. Namely for two non-terminals with the same label, the alphabet of r_1 is often a subset of the alphabet of r_2. Hence, we only need to check whether $L(r_1) \cap L(r_2)$ is empty in most cases, which is more efficient.

2.2 Terms and Subtype Relation

Checking the subtype relations between type expressions is the primary task of type checking. Given the alphabet $V_N \cup V_T$, *terms* T and their root nodes $rt(T)$ are defined as follows:

Definition		$rt(T)$
$T ::= N$	$N \in V_N$	$\{N\}$
ϵ	empty term	$\emptyset$
$l(T_1)$	$l \in V_T$	$\{l\}$
T_1, T_2	concatenation	$rt(T_1) \cup rt(T_2)$
$T_1 \mid T_2$	union	$rt(T_1) \cup rt(T_2)$
T_1*	repetition	$rt(T_1)$

For the other regular expression operators "?" and "+", $T?$ equals to $T|\epsilon$; $T+$ equals to $T, T*$.

According to the definition, each term is actually in the tree structure, describing a sequence of tree nodes. The internal nodes of a term are labeled with $l, l \in V_T$ and the other nodes, called the leaf nodes, are labeled with ϵ or $N, N \in V_N$. For $l(T_1)$, each root node of T_1 is a child node of l and l is the parent node.

If an RRTG $G = (V_N, V_T, N_S, P)$ is applied to XML type checking, each term over the alphabet $V_N \cup V_T$ can be regarded as a type expression in subtyping. The subtype relation between two terms T_1 and T_2 is defined semantically [18]: T_1 is a subtype of T_2, denoted by $T_1 <: T_2$, if each element of $\tau(T_1)$ belongs to $\tau(T_2)$, that is, $T_1 <: T_2 \Leftrightarrow \tau(T_1) \subseteq \tau(T_2)$.

2.3 Other Notions

Given an RRTG $G = (V_N, V_T, N_S, P)$ in the form mentioned above, let $Rhs(N)$ denote the right-hand side of the production rule with the left-hand side N. Given three terms T, X and Y over the alphabet $V_N \cup V_T$, let Σ_T denote the non-terminals occurring in T, $T(X/Y)$ denote the replacement of X in T with Y, $X \prec Y$ denote that X is a subexpression of Y.

Definition 3. *Given a term T, the depth of each root node of T is 1, and the depth of a node s, denoted by $dp(s)$, is $k+1$ if $dp(s') = k$ and s is a child node of s'. The depth of T, denoted by $Dp(T)$, is the maximal depth of all nodes in T. $Dp(\epsilon) = 0$.*

Note that $dp(s)$ may have different values if more than one nodes are labeled with the same symbol s in a term. So we specify the node s before using $dp(s)$ to make sure that no confusion is caused.

Our method works in a bottom-up way on the structures of terms. Hence we define the bottom expressions of a term.

Definition 4. *The bottom expression set of a term T, denoted by $bes(T)$, is defined as follows:*

$$bes(T) = \begin{cases} \emptyset, & if\ Dp(T) = 0 \\ \{T\}, & if\ Dp(T) = 1 \\ \{l_i(T_i) \mid l_i(T_i) \prec T, dp(l_i) = Dp(T) - 1\}, & if\ Dp(T) > 1 \end{cases}$$

We can see that l_i is one of the deepest label of T if $l_i(T_i) \in bes(T)$. For instance, $bes(store(dvd(Title, Price), Dvd_2)) = \{dvd(Title, Price)\}$

In our method, subtyping two terms is based on checking inclusion between regular expressions. Hence we define the following notions to help explain how to obtain the appropriate regular expressions from terms.

Definition 5. *The unfolding tree of a non-terminal N, denoted by $ut(N)$, is defined as follows:*

$$ut(N) = \begin{cases} N, & if\ \Sigma_{Rhs(N)} = \emptyset \\ N(ut(N_1), ..., ut(N_k)), & if\ \Sigma_{Rhs(N)} = \{N_1, ..., N_k\} \subseteq V_N \end{cases}$$

Assume that N' is a non-terminal appearing in $ut(N)$, denoted by $N' \in ut(N)$. The unfolding length of N' with respect to N, denoted by $ufl^N(N')$, is defined as follows, where min indicates choosing the minimal element of a set.

$$ufl^N(N') = \begin{cases} 0, & if\ N = N' \\ 1 + min\{ufl^{N_i}(N') \mid N_i \in \Sigma_{Rhs(N)}, N' \in ut(N_i)\}, & if\ N \neq N' \end{cases}$$

Note that $ufl^N(N')$ is meaningless if N' does not appear in $ut(N)$. An unfolding tree may be infinite for a recursive grammar. However, only the unfolding lengths are concerned. The unfolding length of a non-terminal N' with respect to N is unique if N is specified.

For instance, $ut(Regulars) = Regulars(Dvd_1(Title, Price))$, from which we can obtain some unfolding lengths such as $ufl^{Regulars}(Price) = 2$.

Definition 6. *A non-terminal N is called an ancestor of a term X, denoted by $N \downarrow X$, if there exists some term Y satisfying $N \Rightarrow^* Y$ and $X \prec Y$.*

For instance, $Regulars \Rightarrow^* dvd(Title, Price)*$, $Discounts \downarrow Dis$, etc.

Definition 7. *Given two regular expressions r_1, r_2 over V_N, suppose $r_1 \neq r_2$ and $r_1 \Rightarrow^* r_2$, a non-terminal A is called a local recursive type (LRT) of $r_1 \Rightarrow^* r_2$, if $A \in \Sigma_{r_1}$ and there exists a regular expression r'_2 satisfying $r'_2 \prec r_2, A \in \Sigma_{r'_2}, A \Rightarrow^* r'_2$, and $r'_2 \neq A$.*

Note that LRTs are non-terminals recursively used in the scope of a label. For instance, A is an LRT of $A \Rightarrow^* B, A|\epsilon$ if $A \rightarrow B, A|\epsilon \in P$, and A is not an LRT of $A \Rightarrow^* l(B, A|\epsilon)$ if $A \rightarrow l(B, A|\epsilon) \in P$ though A is recursively defined.

Definition 8. *Given two regular expressions r_1 and r_2, we say that the pair (r_1, r_2) is comparable if $\Sigma_{r_1} \subseteq \Sigma_{r_2}$.*

For instance, (Dvd_1, Dvd_1*) is comparable and $(Dvd_1, Regulars)$ is not.

3 Subtyping

3.1 Comparable Regular Expressions

According to the structures of terms, the inclusion relations between regular expressions may be checked at each level in a bottom-up process of our method. Sometimes the inclusion relations can not be checked directly. For two regular expressions r_1 and r_2 over V_N, $L(r_1) \subseteq L(r_2)$ may not be satisfied if $r_1 <: r_2$. For instance, using G_1 from Example 1, $Dvd_1 <: Regulars$. However, this subtype relation can not be derived from checking inclusion between the regular expressions Dvd_1 and $Regulars$ because they are not comparable as regular expressions. To check this subtype relation by testing inclusion of the regular expressions, first we should replace $Regulars$ with an appropriate regular expression derived from it. Here is Dvd_1*.

Given two regular expressions r and r' over V_N with $r \Rightarrow^* r'$, the regular expression r' is *appropriate* for a non-negative integer m if, given a non-terminal N such that $N \downarrow r$ and $N \downarrow r'$, r' does not contain a non-terminal N_i which satisfies both the following conditions: (1) $\mathit{ufl}^N(N_i) < m$, (2) $Rhs(N_i)$ is a regular expression over V_N.

Note that r' is a regular expression over V_N and r' may be different for different m. How to choose m will be introduced below. Here the appropriate regular expression derived from $Regulars$ is Dvd_1* for $N = Store, m = 2$. And then (Dvd_1, Dvd_1*) is comparable so inclusion can be checked.

The following algorithm $comp(N, r_1, r_2)$ transforms (r_1, r_2) to (r'_1, r'_2) which is comparable, where r'_1 and r'_2 are the appropriate regular expressions derived from r_1 and r_2 respectively. In the algorithm, the non-terminal N is obtained by using the information of label (see Section 3.3). And N is a common ancestor of r_1 and r_2, which is a premise of this algorithm.

Algorithm 1. *comp*(N, r_1, r_2)

```
    1. $U = \{ufl^N(N_i) \mid N_i \in \Sigma_{r_1}\}$
    2. $V = \{ufl^N(N_i) \mid N_i \in \Sigma_{r_2}\}$
    3. $m$ is the maximal element in $U \cup V$
    4. $r'_1 = unfd(N, r_1, m)$
    5. $r'_2 = unfd(N, r_2, m)$
    6. $r'_1 := r'_1(N_1/\epsilon)$ for each term $N_1$ if $N_1 \Rightarrow^* \epsilon$
    7. $r'_2 := r'_2(N_2/\epsilon)$ for each term $N_2$ if $N_2 \Rightarrow^* \epsilon$
    8. return $(r'_1, r'_2)$
```

In Algorithm 1, the function $unfd(N, r, m)$ calculates an appropriate regular expression r' derived from r for the integer m and eliminates LRTs therein. According to the integer m and the unfolding lengths of non-terminals in r with respect to N, the order of deriving of $r \Rightarrow^* r'$ is controlled in this function. Meanwhile, we design a strategy of eliminating LRTs. Assume that $r \Rightarrow^* r_1 \Rightarrow^* r_2 \Rightarrow^* r'$ and A is an LRT of $r_1 \Rightarrow^* r_2$, there must be a subexpression r'_2 of r_2 such that

$$A \Rightarrow^* r'_2 = (R_1, A) \mid ... \mid (R_n, A) \mid R_{n+1} \ ,$$

where each R_i is a regular expression over V_N, and $A \notin \Sigma_{R_i}$. We briefly explain why r'_2 is in the above form: First note that both r_1 and r_2 are regular expressions over V_N; second, in each branch (R_i, A), non-terminal A must occur in the tail position because the recursive uses of non-terminals must occur in the tail positions of content models; and there must be a branch not containing A, here R_{n+1}. Otherwise A can not derive a string which does not contain any non-terminal, namely it is not well defined. So we can get

$$A \textit{ equals to } (R_1 \mid ... \mid R_n)*, R_{n+1} \ .$$

And since $A \Rightarrow^* r'_2$ is known, the LRT A can be eliminated in r_2 by replacing the whole subexpression r'_2 with $(R_1 \mid ... \mid R_n)*, R_{n+1}$, that is,

$$r_2 := r_2(r'_2 \ / \ ((R_1 \mid ... \mid R_n)*, R_{n+1})) \ .$$

Though some LRTs are eliminated from r to r', we still say that r' is an expression derived from r for succinctness, also denoted by $r \Rightarrow^* r'$.

Besides, the strategy of eliminating LRTs can make subtyping union types easier. The main difficulty of subtyping arises from union types [19]. Considering checking $l(A, B) <: l(C, D)|l(E, F)$. The following rule may be used.

$$\frac{l(A, B) <: l(C, D) \quad or \quad l(A, B) <: l(E, F)}{l(A, B) <: l(C, D)|l(E, F)}$$

However, this rule is too weak. For instance, neither premise holds for checking $l((C|E), D) <: l(C, D)|l(E, D)$. Another solution is one that transforms the left-hand side to $l(C, D)|l(E, D)$ by distributing all union types over labels. However, this solution does not work for recursive types, where the distributivity may be infinite. In our method, LRTs are the recursive types which essentially influence the distributivity. Other recursive types which are not LRTs do not affect the distributivity because only the inclusion of regular expressions at the bottom level is considered in bottom-up process. Since LRTs are eliminated in the function $\mathit{unfd}(N, r, m)$, the distributivity of union types over labels can be applied finitely in our method. And then the subtype relations can reduce to checking the subtype relation of each branch of union types.

Algorithm 1 can be illustrated by Figure 1, where each triangle denotes the unfolding tree $ut(N)$, the solid lines inside of triangles denote regular expressions over V_N, the distances between the tops of triangles and the solid lines denote the unfolding lengths of the non-terminals occurring in the regular expressions with respect to N, the broken lines show the bound of non-terminals occurring in regular expressions. Before executing Algorithm 1, the leftmost triangle shows that the maximal unfolding length is m and $N \downarrow r_1, N \downarrow r_2$. The other two triangles express the results of Algorithm 1. The unfolding lengths of all non-terminals in r_1' and r_2' with respect to N are close to m as far as possible. As shown in Figure 1, the solid lines of r_1' and r_2' are not straight, meaning that r_1' and r_2' may contain some non-terminals whose unfolding lengths with respect to N are less than m. Of course, the right-hand sides of the production rules of these non-terminals are definitely contain some label.

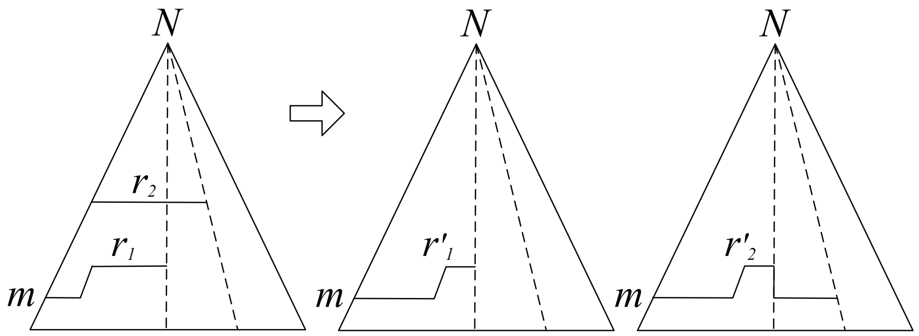


Fig. 1. Illustration of $\mathit{comp}(N, r_1, r_2)$

Example 2. Assume that the production rule (2) in G_1 is replaced with the recursive one:

$$\mathit{Regulars} \rightarrow (\mathit{Dvd}_1, \mathit{Regulars})|\epsilon$$

Let $(r_1, r_2) = (\mathit{Dvd}_1, \mathit{Regulars}), N = \mathit{Store}$. We can have that $\mathit{Regulars}$ is the LRT of $\mathit{Regulars} \Rightarrow (\mathit{Dvd}_1, \mathit{Regulars})|\epsilon$. So $\mathit{Regulars}$ equals to Dvd_1* and $\mathit{comp}(\mathit{Store}, \mathit{Dvd}_1, \mathit{Regulars}) = (\mathit{Dvd}_1, \mathit{Dvd}_1*)$.

Lemma 1. *Given two regular expressions r_1, r_2 over V_N with $r_1 <: r_2$, if there is a non-terminal N, such that $N \downarrow r_1, N \downarrow r_2$, and $(r_1', r_2') = \mathit{comp}(N, r_1, r_2)$, then (r_1', r_2') is comparable.*

Proof. Assume that (r'_1, r'_2) is not comparable. There must be at least one non-terminal N' such that $N' \in \Sigma_{r'_1}$ and $N' \notin \Sigma_{r'_2}$. Since the restriction of RRTGs, it will immediately lead to a contradiction to $r_1 <: r_2$ if N' does not occur in any expression derived from r_2. So N' must occur in some expression derived from r_2, denoted by r''_2, satisfying one of the following two cases:

$$(1)\ r_2 \Rightarrow^* r''_2 \Rightarrow^* r'_2 \quad (2)\ r_2 \Rightarrow^* r'_2 \Rightarrow^* r''_2$$

For case (1), $\mathit{ufl}^N(N')$ is as close to m as possible since $N' \in \Sigma_{r'_1}$. So N' must occur in r'_2. This leads to a contradiction. For case (2), there must be two non-terminals N_2 and N'_2 satisfying $N_2 \in \Sigma_{r_2}, N'_2 \in \Sigma_{r'_2}$ and $N \downarrow N_2 \downarrow N'_2 \downarrow N'$ so we can get $\mathit{ufl}^N(N') = \mathit{ufl}^N(N_2) + \mathit{ufl}^{N_2}(N'_2) + \mathit{ufl}^{N'_2}(N')$. And $\mathit{ufl}^N(N') \leq m$ since $N' \in \Sigma_{r'_1}$. So $\mathit{ufl}^N(N_2) + \mathit{ufl}^{N_2}(N'_2) < m$, that is $\mathit{ufl}^N(N'_2) < m$. But now N'_2 can continue deriving a new expression because r''_2 exists. This leads to a contradiction to the notion of appropriate regular expressions. Both cases can not be satisfied. □

Lemma 2. *Given two regular expressions r_1, r_2 over V_N, if there exists a non-terminal $N \in V_N$, such that $N \downarrow r_1, N \downarrow r_2$, and $(r'_1, r'_2) = comp(N, r_1, r_2)$, then $L(r'_1) \subseteq L(r'_2)$ if and only if $r_1 <: r_2$.*

Proof. We need to prove the following two cases:

$$(1)\ L(r'_1) \subseteq L(r'_2) \Rightarrow r_1 <: r_2 \qquad (2)\ r_1 <: r_2 \Rightarrow L(r'_1) \subseteq L(r'_2)$$

(1): $L(r'_1) \subseteq L(r'_2)$ means $\tau(r'_1) \subseteq \tau(r'_2)$. Immediately we can obtain $\tau(r'_1) = \tau(r_1)$ and $\tau(r'_2) = \tau(r_2)$ since $r_1 \Rightarrow^* r'_1$ and $r_2 \Rightarrow^* r'_2$. So $\tau(r_1) \subseteq \tau(r_2)$, namely $r_1 <: r_2$. (2): First we can get $r'_1 <: r'_2$ since $r_1 \Rightarrow^* r'_1$ and $r_2 \Rightarrow^* r'_2$. And then by Lemma 1, (r'_1, r'_2) is comparable. So $L(r'_1) \subseteq L(r'_2)$. □

Lemma 1 and Lemma 2 show the correctness of Algorithm 1. The subtype relation between regular expressions r_1 and r_2 can reduce to checking inclusion between r'_1 and r'_2, which are the appropriate regular expressions derived from them.

3.2 Subtyping Algorithm

Given an RRTG $G = (V_N, V_T, N_S, P)$ and two terms T_1 and T_2 over $V_N \cup V_T$, the aim of subtyping is to check $T_1 <: T_2$. Assume that T_2 is a regular expression over V_N for simplicity; this assumption does not lose generality because if T_2 contains some label, we can replace T_2 in the subtyping algorithm with a new non-terminal N and construct an auxiliary RRTG, where N is the start symbol and $\tau(T_2) = \tau(N)$. In most cases, the auxiliary RRTG can be easily constructed. For instance, there is only one rule of the form $N \rightarrow l(r)$ in the auxiliary RRTG if $T_2 = l(r)$. The production rules in the auxiliary RRTG will have higher priority to be used in checking $T_1 <: N$ so that the disjointness is not required between any two non-terminals from the auxiliary RRTG and the original grammar respectively. Accordingly, the auxiliary RRTG can be constructed easily.

The subtyping algorithm $subty(T_1, T_2)$ works in a bottom-up way on the structures of terms. T_1 is reduced during the procedure, which resembles the reduction of the production rules of grammars. For instance, as a bottom expression of T_1, $l_i(T_i)$ will be reduced to a term T in T_1 if the subtype relation $l_i(T_i) <: T$ is satisfied. Then the process goes to upper level. The subtyping algorithm is as follows. For each $l_i(T_i) \in bes(T_1)$, first eliminate LRTs in it and distribute union types over labels in T_i if necessary. Then calculate $T_1(l_i(T_i)/T)$ if the subtype relation $l_i(T_i) <: T$ is satisfied, which reduces to checking inclusion between regular expressions. At last, T_1 may be reduced to a regular expression over V_N so inclusion between this regular expression and T_2 will be checked finally.

Algorithm 2. *subty*(T_1, T_2)

```
1. while $Dp(T_1) > 1$
2.     find a bottom expression $l_i(T_i) \in bes(T_1)$
3.     if $l_i(T_i) <: T$ is satisfied
4.         $T_1 := T_1(l_i(T_i)/T)$
5. end while
6. check $T_1 <: T_2$
```

More concretely, by using the judgement $\Gamma \vdash T_1 <: T_2 \Rightarrow \Gamma'$, similar with XDuce [2], which is read "If all subtype relations in the input set Γ hold, then $T_1 <: T_2$ is satisfied and added to the output set Γ', which includes all old subtype relations in Γ and new subtype relations that have been checked in the process", the subtyping algorithm can be expressed by the following rules of the form

$$\frac{\begin{array}{c} x \\ \Gamma \vdash y_1 \Rightarrow \Gamma_1 \quad \Gamma_1 \vdash y_2 \Rightarrow \Gamma_2 \quad ... \quad \Gamma_{k-1} \vdash y_k \Rightarrow \Gamma_k \end{array}}{\Gamma \vdash z \Rightarrow \Gamma_k},$$

which is read "given x, if $y_1, y_2, ..., y_k$ can be achieved, then z is satisfied".

In the following rules, we write $L(r_1) \subseteq L(r_2)$ instead of $r_1 <: r_2$ in order to highlight the using of inclusion checking of regular expressions.

$$\frac{\begin{array}{c} Dp(T_1) \leq 1,\ (r_1, r_2) = comp(N_S, T_1, T_2) \\ \Gamma \vdash L(r_1) \subseteq L(r_2) \Rightarrow \Gamma' \end{array}}{\Gamma \vdash T_1 <: T_2 \Rightarrow \Gamma'} \tag{1}$$

$$\frac{\begin{array}{c} Dp(T_1) > 1,\ l_i(T_i) \in bes(T_1) \\ \Gamma \vdash l_i(T_i) <: T \Rightarrow \Gamma' \quad \Gamma' \vdash T_1(l_i(T_i)/T) <: T_2 \Rightarrow \Gamma'' \end{array}}{\Gamma \vdash T_1 <: T_2 \Rightarrow \Gamma''} \tag{2}$$

Rule (1): $Dp(T_1) \leq 1$, namely T_1 is an empty term or a regular expression over V_N. The subtype relation reduces to checking inclusion between r_1 and r_2, where (r_1, r_2) is comparable corresponding to (T_1, T_2).

Rule (2): $Dp(T_1) > 1$. $T_1 <: T_2$ if $l_i(T_i) <: T$ and $T_1(l_i(T_i)/T) <: T_2$, where $l_i(T_i) <: T$ is checked by using the following rules (3)(4)(5), corresponding to the line 3 of Algorithm 2. And $T_1(l_i(T_i)/T) <: T_2$ makes the whole procedure work in a bottom-up way. Note that the term T is unique because of RRTGs.

In the following rules (3)(4)(5), each non-terminal N and regular expression r in set S are obtained by using the information of label l_i. $N \downarrow T_i$ if $l_i(T_i) <: N$. So N is the non-terminal used in $comp(N, T_i, r)$ to calculate (r_1, r_2). In most cases, as in the rules (3)(4), the term T at line 3 of Algorithm 2 is the non-terminal N. Only in the rule (5) for dealing with union types, T is an union of non-terminals. $dist(l_i(r_1))$ distributes r_1 over l_i to the corresponding union types if r_1 contains some top-level (not occurring in the range of any regular expression operator "$*$") regular expression operator "$|$", the latter situation is denoted by $\mu(r_1)$.

$$\frac{S = \{(N,r) | N \to l_i(r) \in P\},\ |S| = 1,\ (r_1, r_2) = comp(N, T_i, r) \quad \Gamma \vdash L(r_1) \subseteq L(r_2) \Rightarrow \Gamma'}{\Gamma \vdash l_i(T_i) <: N \Rightarrow \Gamma'} \tag{3}$$

$$\frac{S = \{(N,r) | N \to l_i(r) \in P\},\ |S| > 1,\ (r_1, r_2) = comp(N, T_i, r),\ \neg\mu(r_1) \quad \Gamma \vdash L(r_1) \subseteq L(r_2) \Rightarrow \Gamma'}{\Gamma \vdash l_i(T_i) <: N \Rightarrow \Gamma'} \tag{4}$$

$$\frac{S = \{(N,r) | N \to l_i(r) \in P\},\ |S| > 1,\ (r_1, r_2) = comp(N, T_i, r),\ \mu(r_1) \quad \Gamma \vdash dist(l_i(r_1)) <: T \Rightarrow \Gamma'}{\Gamma \vdash l_i(T_i) <: T \Rightarrow \Gamma'} \tag{5}$$

Rule (3): $|S| = 1$. It means that the non-terminals and labels are in one-to-one correspondence. The non-terminal N and the content model r are unique. $l_i(T_i) <: N$ if $T_i <: r$, which reduces to checking inclusion between r_1 and r_2.

Rule (4): $|S| > 1$ and $\neg\mu(r_1)$. The process finds if there is an element (N, r) in S satisfying: $(r_1, r_2) = comp(N, T_i, r)$ such that $L(r_1) \subseteq L(r_2)$. In this rule, as mentioned above, the production rules in the auxiliary RRTG will have higher priority to be used to obtain the non-terminal N and the content model r. It means that the elements of set S are actually divided into two classes with different priorities. Note that r is unique because of RRTGs and the priorities.

Rule (5): $|S| > 1$ and $\mu(r_1)$. $l_i(T_i) <: T$ if $dist(l_i(r_1)) <: T$, which reduces to checking each branch of $dist(l_i(r_1))$. Here we use the following rule (*union*), where each T_i is a branch of $dist(l_i(r_1))$.

$$\frac{T_1 <: T_1' \ , ..., \ T_n <: T_n' \quad and \quad T = T_1'|...|T_n'}{T_1|...|T_n <: T} \tag{union}$$

The above rules can be simplified greatly if each production rule is of the form $N \to lr$, corresponding to the tree grammars in normal form [16], since each pair (T_i, r) to be checked is already comparable. If the RRTG G is actually a DTD, then the set S contains only one element so that rules (1)(2)(3) are sufficient and subtyping will be more efficient. So the subtyping algorithm is not time-consuming in practice since most actually used XML Schemas are actually DTDs [17]. Because the algorithm is based on checking inclusion between regular expressions, the complexity of this algorithm depends on the regular expressions used in XML schema languages. It is PSPACE for general regular expressions. And it will be more efficient if some special regular expressions which have lower complexity for inclusion testing are used, such as one-unambiguous regular expressions with the complexity PTIME, which are recommended to use in DTDs and XML Schemas by W3C. Two inclusion algorithms of one-unambiguous regular expressions have been proposed in [20].

Example 3. Using the grammar G_1 in Example 1, for the following two terms,

$$T_1 = store(Dvd_1, dvd(Title, Price, (Dis|\epsilon)))$$
$$T_2 = Store$$

the procedure of checking $T_1 <: T_2$ is illustrated in Figure 2, where $Title, Price$ and Dis are abbreviated to T, P and D.

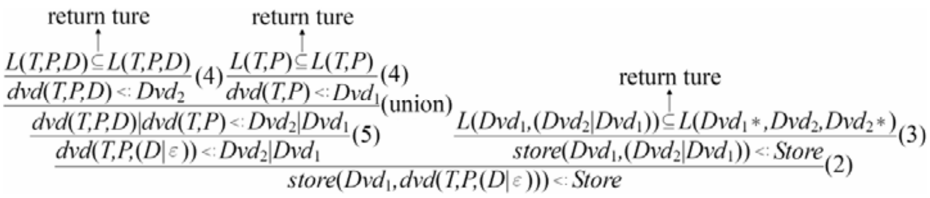

return ture
return ture
return ture
L(T,P,D) ⊆ L(T,P,D) (4)
dvd(T,P,D) <: Dvd₂
L(T,P) ⊆ L(T,P) (4)
dvd(T,P) <: Dvd₁
(union)
dvd(T,P,D)|dvd(T,P) <: Dvd₂|Dvd₁ (5)
dvd(T,P,(D|ε)) <: Dvd₂|Dvd₁
L(Dvd₁,(Dvd₂|Dvd₁)) ⊆ L(Dvd₁*,Dvd₂,Dvd₂*) (3)
store(Dvd₁,(Dvd₂|Dvd₁)) <: Store (2)
store(Dvd₁,dvd(T,P,(D|ε))) <: Store

Fig. 2. Example of subtyping

The inclusion checking of regular expressions are called three times and all return *true* in this procedure. However, in the previous two times, the two regular expressions are the same so that the inclusion algorithm will return *true* directly. So the inclusion checking of regular expressions is actually executed only one time in this procedure.

Theorem 1. *The above subtyping rules for RRTGs are correct.*

Theorem 1 can be proved easily by using the semantics of types, that is, $T_1 <: T_2$ if and only if $\tau(T_1) \subseteq \tau(T_2)$, and the transitivity of subtype relations, which can be proved by the set theory. The details are omitted here.

For instance, we briefly prove the subtyping rule (4): If $L(r_1) \subseteq L(r_2)$ is satisfied, we can get $r_1 <: r_2$. Since r_1, r_2 are the appropriate regular expressions derived from T_i and r respectively, so $T_i <: r$. Then immediately $l_i(T_i) <: l_i(r)$. Since $N \rightarrow l_i(r) \in P$ is known, we get the result $l_i(T_i) <: N$.

4 Experiments

We have implemented the above subtyping method and accomplished some experiments to compare with XDuce. Our concern is the times of type checking of test cases by using XDuce and our method. The results are shown in Table 1. Column t shows the time of type checking of XDuce. Column t' shows the time of our method, which includes the time of pretreatment, such as transforming

Table 1. Results of experiments (time: seconds)

test cases	t	t'	*ratio*	s_1	s_2
addrbook	0.003188	0.001775	56%	67	-
bookmarks	0.581173	0.170046	29%	216	1198
html2latex1	1.204572	0.319842	27%	300	1198
html2latex2	1.175982	0.365335	31%	236	1198
ns2xbel	0.002295	0.001465	64%	77	94
rng2xduce	0.613078	0.258772	42%	451	-
polysample	0.002003	0.001206	60%	83	-
dvdstore	0.532574	0.327925	62%	35	-

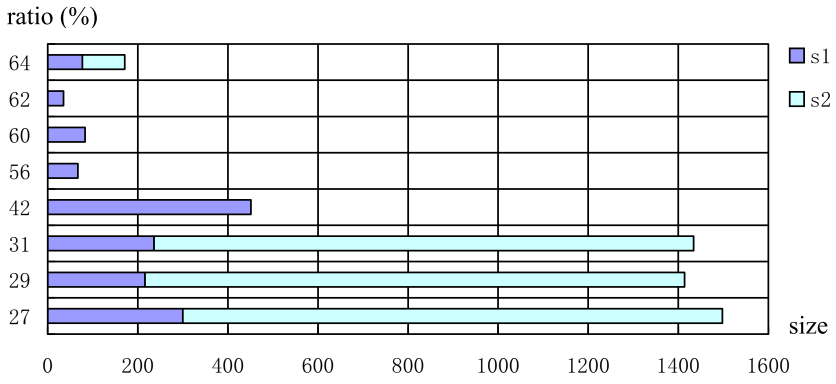


Fig. 3. Ratios and sizes

the type definitions of test cases into the form of RRTGs. Column *ratio* is the ratio of t' to t. The last two columns show the sizes of test cases and external DTDs (may not exist, denoted by "-"), measured by the number of lines of codes.

Most test cases are from the XDuce distribution [21]. The type definitions of test cases except the last one are all DTDs. These test cases basically cover the actually used scenarios of XML processing, including fragment extraction, type conversion and so on. Some test cases are small in size and some are not.

1) *addrbook* defines a DTD of an address book file and extracts a phone book.
2) *bookmarks* converts a Netscape bookmark file (a subset of type HTML) into a file of full HTML type with an external DTD (xhtml1-transitional.dtd).
3) *html2latex1* converts a file (of type HTML) into a LaTeX file (of type string) with an external DTD (xhtml1-transitional.dtd).
4) *html2latex2* is similar to *html2latex1*, from the CDuce benchmarks [22].
5) *ns2xbel* converts a Netscape bookmark file into XBEL format with an external DTD (xbel-1.0.dtd).
6) *rng2xduce* converts a file of type Relax NG into XDuce format.
7) *polysample* is a simple polymorphic program.

8) *dvdstore* is a test case written by ourselves. Similar to *addrbook*, the main function is to extract data from the input file. However, the type definition of *dvdstore* equals to a RRTG, which has been given in Example 1.

The results show that our method is more efficient than XDuce. The time required for type checking is about halved on average. We also analyze the test cases and results of our method. Currently, it seems that the effect of our method is more evident when the test case is larger in size, as shown in Figure 3, where each histogram is corresponding to a test case from Table 1 and s_1, s_2 express the same meanings as in Table 1. Of course the experiments we did are still preliminary. More test cases would be very useful and the implementation of our method can still be improved.

The environment is as follows: Intel Pentium 4 CPU 3.0GHz, 256MB RAM, Ubuntu 7.10, XDuce 0.5.0, OCaml 3.10.0.

5 Concluding Remarks

We have proposed regular tree grammars with disjoint production rules and a new subtyping method for the restricted tree grammars. Differing from the tree-automata based methods, our subtyping method reduces to checking inclusion between regular expressions together with using information of labels in the grammars. The method works in a bottom-up way on structures of type expressions. In addition, the inclusion algorithms of regular expressions can be chosen depending on the classes of regular expressions used in XML schema languages, which may have different complexities. We believe that our method will be more efficient if some special regular expressions are used in XML schema languages, such as one-unambiguous regular expressions.

There are several future works. Though most actually schemas fall into RRTGs, the cases in which non-RRTGs (e.g., the grammar G_2 mentioned in Section 2.1) can be easily transformed to RRTGs still deserve study. However, the transformation of grammars may be difficult when the grammars are complex. We will consider a strategy which combines the top-down way with the bottom-up way in subtyping. Optimizations and integration with other methods are also concerned.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful reviews. The work is supported by the National Natural Science Foundation of China under Nos. 60573013, 60721061.

References

1. Hosoya, H., Pierce, B.C.: XDuce: A statically typed XML processing language. ACM Trans. Interet Technol. 3(2), 117–148 (2003)
2. Hosoya, H., Vouillon, J., Pierce, B.C.: Regular expression types for XML. ACM Trans. Program. Lang. Syst. 27(1), 46–90 (2005)

3. Benzaken, V., Castagna, G., Frisch, A.: CDuce: an XML-centric general-purpose language. In: ICFP, pp. 51–63 (2003)
4. Sulzmann, M., Lu, K.Z.M.: XHaskell - adding regular expression types to Haskell. In: Chitil, O., Horváth, Z., Zsók, V. (eds.) IFL 2007. LNCS, vol. 5083, pp. 75–92. Springer, Heidelberg (2008)
5. Hosoya, H., Pierce, B.C.: Regular expression pattern matching for XML. J. Funct. Program 13(6), 961–1004 (2003)
6. Frisch, A.: Regular tree language recognition with static information. In: IFIP TCS, pp. 661–674 (2004)
7. Suzuki, N.: An edit operation-based approach to the inclusion problem for DTDs. In: SAC, pp. 482–488 (2007)
8. Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E., Yergeau, F.: Extensible markup language (XML) 1.0 (2008), `http://www.w3.org/tr/xml`
9. Champavère, J., Gilleron, R., Lemay, A., Niehren, J.: Efficient inclusion checking for deterministic tree automata and DTDs. In: Martín-Vide, C., Otto, F., Fernau, H. (eds.) LATA 2008. LNCS, vol. 5196, pp. 184–195. Springer, Heidelberg (2008)
10. Martens, W., Neven, F., Schwentick, T.: Complexity of decision problems for simple regular expressions. In: Fiala, J., Koubek, V., Kratochvíl, J. (eds.) MFCS 2004. LNCS, vol. 3153, pp. 889–900. Springer, Heidelberg (2004)
11. Martens, W., Neven, F., Schwentick, T., Bex, G.J.: Expressiveness and complexity of XML Schema. ACM Trans. Database Syst. 31(3), 770–813 (2006)
12. Ghelli, G., Colazzo, D., Sartiani, C.: Efficient inclusion for a class of XML types with interleaving and counting. In: Arenas, M., Schwartzbach, M.I. (eds.) DBPL 2007. LNCS, vol. 4797, pp. 231–245. Springer, Heidelberg (2007)
13. Colazzo, D., Ghelli, G., Sartiani, C.: Efficient asymmetric inclusion between regular expression types. In: ICDT, pp. 174–182 (2009)
14. Sperberg-McQueen, C., Thompson, H.: XML Schema (2005), `http://www.w3.org/XML/Schema`
15. Brüggemann-Klein, A., Wood, D.: One-unambiguous regular languages. Inf. Comput. 142(2), 182–206 (1998)
16. Murata, M., Lee, D., Mani, M., Kawaguchi, K.: Taxonomy of XML schema languages using formal language theory. ACM Trans. Internet Techn. 5(4), 660–704 (2005)
17. Bex, G.J., Neven, F., den Bussche, J.V.: DTDs versus XML Schema: a practical study. In: WebDB, pp. 79–84 (2004)
18. Castagna, G., Frisch, A.: A gentle introduction to semantic subtyping. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 30–34. Springer, Heidelberg (2005)
19. Vouillon, J.: Subtyping union types. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 415–429. Springer, Heidelberg (2004)
20. Chen, H., Chen, L.: Inclusion test algorithms for one-unambiguous regular expressions. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 96–110. Springer, Heidelberg (2008)
21. XDuce source, `http://sourceforge.net/projects/xduce`
22. CDuce benchmarks, `http://www.cduce.org/bench.html#xduce`

Minimal Tree Language Extensions: A Keystone of XML Type Compatibility and Evolution[⋆]

Jacques Chabin[1], Mirian Halfeld-Ferrari[1], Martin A. Musicante[2], and Pierre Réty[1]

[1] Université d'Orléans, LIFO, Orléans, France
{jacques.chabin,mirian,pierre.rety}@univ-orleans.fr
[2] Universidade Federal do Rio Grande do Norte, DIMAp Natal, Brazil
mam@dimap.ufrn.br

Abstract. In this paper, we propose algorithms that extend a given regular tree grammar G_0 to a new grammar G respecting the following two properties: (*i*) G belongs to the sub-class of local or single-type tree grammars and (*ii*) G is the *least grammar* (in the sense of language inclusion) that contains the language of G_0. Our algorithms give rise to important tools in the context of web service composition or XML schema evolution. We are particularly interested in applying them in order to reconcile different XML type messages among services. The algorithms are proven correct and some of their applications are discussed.

1 Introduction

When dealing with web service composition, one should consider the problem of how to reconcile structural differences among types of XML messages supported by different services. A web service designer may wish to implement a service S that is able to accept XML messages coming from different services A, B or C (*i.e.* services offering the same service, in slightly different formats). To allow the composition of S with any of these services, it would be practical to infer a *general* type from the message types of services A, B and C. Moreover, the initial type accepted by S may evolve if one decides to consider also messages coming from a new service D.

The learning of new types (or *schema*) can be very helpful for the harmonious work of the applications that manipulate these data. Some algorithms for learning XML data have been proposed [GGR+00, Chi01, BNV07]. In general, these algorithms consist of learning the schema using sets of (positive or negative) examples. Our work considers another situation, since we aim at integrating different schemas in order to implement a service or to adapt it to a new environment: our goal is to maintain the global behavior of a composition while extending the type of the messages being processed.

We are thus interested in automatically generating a new type which is a conservative extension of some given types. Moreover, we are interested in obtaining

⋆ Supported by: Codex ANR-08-DEFIS-04, e-cloudss (LACCIR), CAPES 3164/08-0.

the *least* schema (in the sense of type inclusion) that complies with this condition and that can be specified in current XML schema language standards such as DTD or XMLSchema. The following example illustrates schema evolution as proposed by our method.

Example 1. Consider a web service, built for a library consortium, capable of giving information about publications existing in different libraries. This service should be able to accept messages from different library services, each of them specified over its own message format. Let G_A and G_B be the DTDs (Local Tree Grammars) which define the type of messages coming from library services A and B respectively. As usual we represent non terminals by starting with a capital letter and terminals with a small letter. Besides the production rules presented below (C and Z are the start symbols), we suppose that all the production rules $X \to x[\epsilon]$ such that (respectively) $X \in \{A, T, D, P, N, V, E, W\}$ and $x \in \{author, title, datePublication, price, number, volume, editor, status\}$ are included in the sets of rules of the schema (grammar):

Productions rules of G_A
$C \to catalog[B^*]$
$B \to book[A^+.T.D.W]$

Productions rules of G_B
$Z \to catalog[(Y \mid L)^*]$
$Y \to book[A^+.T.D.P?.E]$
$L \to article[A^+.T.D.J.E]$
$J \to journal[N.V]$

The simple union of these two grammars ($G_0 = G_A \cup G_B$) is not a solution to the problem, since it is *not* a Local Tree Grammar (LTG). This means that it cannot be described as a well-formed DTD. Our algorithm starts processing G_0 and returns the following grammar G_2 as output (by merging some production rules of G_0), where C_Z is the start symbol.

Productions rules of G_2	
$C_Z \to catalog[(B_Y \mid L)^* \mid B_Y^*]$	$B_Y \to book[A^+.T.D.W \mid A^+.T.D.P?.E]$
$L \to article[A^+.T.D.J.E]$	$J \to journal[N.V]$

Our algorithms are capable of finding a definition for the *least* (local or single-type) tree language (set of XML documents) that contains the XML documents described by both original types. For instance, in Example 1, grammar G_2 is the least LTG that contains the languages generated by G_A and G_B.

The algorithms proposed in this paper are able to transform a (general) regular tree grammar into a Local or Single-Type tree grammar (the user can choose whether the resulting grammar will be a LTG or a STTG). Our algorithms are proven correct for any regular tree grammar in reduced normal form.

The rest of this paper is organized as follows: in Section 2 we recall the theoretical background needed to the introduction of our method; Section 3 presents our schema evolution algorithms for LTG and STTG and Section 4 discusses the implementation of these methods. The paper finishes by considering some related work and by discussing our perspectives of work.

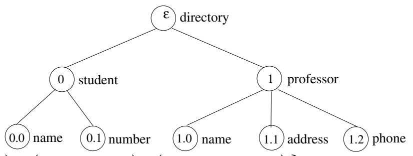
2 Theoretical Background

It is a well known fact that type definitions for XML and regular tree grammars are similar notions and that some schema definition languages can be represented by using specific classes of regular tree grammars. Thus, DTD and XML Schema, correspond, respectively, to Local Tree Grammars and Single-type Tree Grammars [MLMK05]. Given an XML type T and its corresponding tree grammar G, the set of XML documents described by the type T corresponds to the language (set of trees) generated by G.

In this paper we consider a tree language as a set of *unranked* trees. Tree nodes have a label (from a set Σ) and a position (given as a string of integers). Let U be the set of all finite strings of non-negative integers with the empty string ϵ as the identity. In the following definition we assume that $Pos(t) \subseteq U$ is a nonempty set closed under prefixes (*i.e.* , if $u \preceq v$, $v \in Pos(t)$ implies $u \in Pos(t)$).

Definition 1 (Unranked Σ-valued tree t). A nonempty unranked Σ-valued tree t is a mapping $t : Pos(t) \rightarrow \Sigma$ where $Pos(t)$ satisfies: $j \geq 0, uj \in Pos(t), 0 \leq i \leq j \Rightarrow ui \in Pos(t)$. The set $Pos(t)$ is called the set of *positions* of t. We write $t(v) = a$, for $v \in Pos(t)$, to indicate that the Σ-symbol associated to v is a. □

The following figure represents a tree whose alphabet is the set of element names appearing in an XML document.



Given a tree t we denote by $t|_p$ the subtree whose root is at position $p \in Pos(t)$,*i.e.* $Pos(t|_p) = \{s \mid p.s \in Pos(t)\}$ and for each $s \in Pos(t|_p)$ we have $t|_p(s) = t(p.s)$.

For instance, in the figure $t|_0 = \{(\epsilon, student), (0, name), (1, number)\}$, or equivalently, $t|_0 = student(name, number)$.

Given a tree t such that the position $p \in Pos(t)$ and a tree t', we note $t[p \leftarrow t']$ as the tree that results of substituting the subtree of t at position p by t'.

Definition 2 (Sub-tree, Forest). Let L be a set of trees. $ST(L)$ is the set of sub-trees of elements of L, i.e. $ST(L) = \{t \mid \exists u \in L, \exists p \in Pos(u), t = u|_p\}$. A *forest* is a (possibly empty) tuple of trees. For $a \in \Sigma$ and a forest $w = \langle t_1, \ldots, t_n \rangle$, $a(w)$ is the tree defined by $a(w) = a(t_1, \ldots, t_n)$. On the other hand, $w(\epsilon)$ is defined by $w(\epsilon) = \langle t_1(\epsilon), \ldots, t_n(\epsilon) \rangle$, i.e. the tuple of the top symbols of w. □

Definition 3 (Regular Tree Grammar). A *regular tree grammar* (RTG) is a 4-tuple $G = (N, T, S, P)$, where: N is a finite set of *non-terminal symbols*; T is a finite set of *terminal symbols*; S is a set of *start symbols*, where $S \subseteq N$ and P is a finite set of *production rules* of the form $X \rightarrow a\,[R]$, where $X \in N$, $a \in T$, and R is a regular expression over N (We say that, for a production rule, X is the left-hand side, $a\,R$ is the right-hand side, and R is the content model.) □

Definition 4 (Derivation). For a RTG $G = (N, T, S, P)$, we say that a tree t built on $N \cup T$ derives (in one step) into t' iff (i) there exists a position p

of t such that $t|_p = A \in N$ and a production rule $A \to a\,[R]$ in P, and (ii) $t' = t[p \leftarrow a(w)]$ where $w \in L(R)$ ($L(R)$ is the set of words of non-terminals generated by R). We write $t \to_{[p, A \to a\,[R]]} t'$. More generally, a derivation (in several steps) is a (possibly empty) sequence of one-step derivations. We write $t \to_G^* t'$.

The language $L(G)$ generated by G is the set of trees containing only terminal symbols, defined by: $L(G) = \{t \mid \exists A \in S,\ A \to_G^* t\}$. □

Example 2. Let $G = (N, T, \{X\}, P)$, where $P = \{X \to f\,[A^*.B], A \to a, B \to b\}$. A derivation from the start symbol is $X \to_{[X \to f\,[A^*.B]]} f(A, A, B) \to_G^* f(a, a, b)$. Consequently $f(a, a, b) \in L(G)$. □

To produce grammars that generate least languages, our algorithms need to start from grammars in reduced form and (as in [ML02]) in normal form. A *regular tree grammar* (RTG) is said to be in **reduced form** if (i) every non-terminal is reachable from a start symbol, and (ii) every non-terminal generates at least one tree containing only terminal symbols. A regular tree grammar (RTG) is said to be in **normal form** if distinct production rules have distinct left-hand-sides.

Example 3. Given the tree grammar $G_0 = (N, T, S, P_0)$, where $N = \{X, A, B\}$, $T = \{f, a, c\}$, $S = \{X\}$, and $P_0 = \{X \to f\,[A.B],\ A \to a,\ B \to a,\ A \to c\}$. Note that G_0 is in reduced form, but it is not in normal form. The conversion of G_0 into the normal form gives the set $P_1 = \{X \to f\,[(A|C).B],\ A \to a,\ B \to a,\ C \to c\}$. Thus $G_1 = (N \cup \{C\}, T, S, P_1)$ is in reduced normal form. □

The following three definitions come from [MLMK05].

Definition 5 (Competing Non-Terminals). Two different non-terminals A and B (of the same grammar G) are said to be *competing with each other* if (i) a production rule has A in the left-hand side, (ii) another production rule has B in the left-hand side, and (iii) these two production rules share the same terminal symbol in the right-hand side. □

Definition 6 (Local Tree Grammar). A *local tree grammar* (LTG) is a regular tree grammar that does not have competing non-terminals. A *local tree language* (LTL) is a language that can be generated by at least one LTG. □

Note that converting a LTG into normal form produces a LTG as well.

Definition 7 (Single-Type Tree Grammar). A *single-type tree grammar* (STTG) is a regular tree grammar in normal form, where (i) for each production rule, non terminals in its regular expression do not compete with each other, and (ii) starts symbols do not compete with each other. A *single-type tree language* (STTL) is a language that can be generated by at least one STTG. □

In [MLMK05] the expressive power of these classes of languages is discussed. We recall that LTL $\subset$ STTL $\subset$ RTL. Moreover, the LTL and STTL are closed under intersection but not under union; while the RTL are closed under union, intersection and difference.

3 Type Evolution

This section describes our type evolution approach by presenting the main theoretical contributions of our work. The algorithms proposed here take as argument a general regular tree grammar in reduced normal form. They produce a LTG (respectively a STTG) whose language is the least LTL (resp. the least STTL) that contains the language described by the original tree grammar.

The intuitive idea underlying both algorithms is to locate sets of competing non-terminal symbols of the tree grammar, and fix the problem by identifying these non-terminals as the same one.

3.1 Transforming a RTG into a LTG

We consider the problem of obtaining a local tree grammar whose language contains a given tree language. Given a regular tree grammar G_0, we are interested in the definition of the *least* local tree language that contains the language generated by G_0. The new language will be described by a local tree grammar. The algorithm described below obtains a new grammar by transforming G_0. The transformation rules are intuitively simple: every pair of competing non-terminals are transformed into one symbol. We show that this simple transformation of the original grammar yields to a local tree grammar in a finite number of steps.

Now, consider some useful properties of local tree languages. These properties will be used to show the correctness of our grammar-transformation algorithm. Proofs of the properties are omitted here due to the lack of space. They are available in [CHMR09]. The following lemma states that the type of the subtrees of a tree node is determined by the label of its node (i.e. the type of each node is *locally* defined). Recall that $ST(L)$ is the set of sub-trees of elements of L.

Lemma 1. *(see also [PV00, Lemma 2.10]) Let L be a local tree language (LTL). Then, for each $t \in ST(L)$, each $t' \in L$ and each $p' \in Pos(t')$, we have that $t(\epsilon) = t'(p') \implies t'[p' \leftarrow t] \in L$.* □

In the following, we also need a weaker version of the previous lemma:

Corollary 1. *Let L be a local tree language (LTL). Then, for each $t, t' \in ST(L)$, and each $p' \in Pos(t')$, we have that $t(\epsilon) = t'(p') \implies t'[p' \leftarrow t] \in ST(L)$.* □

In practical terms, Corollary 1 gives us a rule of thumb on how to "complete" a regular language in order to obtain a local tree language. For instance, let $L = \{f(a(b), c), f(a(c), b)\}$ be a regular language. According to Corollary 1, we know that L is not LTL and that the least local tree language L' containing L contains all trees where a has c as a child together with all trees where a has b as a child. In other words, $L' = \{f(a(b), c), f(a(c), b), f(a(c), c), f(a(b), b)\}$.

Let us now define our first algorithm for schema (DTD, Local Tree Grammar) evolution. The main intuition behind our algorithm is to merge rules having competing non terminals in their left-hand side. New non terminals are introduced in other to replace competing ones.

Definition 8 (RTG into LTG Transformation). Let $G_0 = (N_0, T_0, S_0, P_0)$ be a regular tree grammar in reduced normal form. We define a new regular tree grammar $G = (N, T, S, P)$, obtained from G_0, according to the following steps:

1. Let $G_2 := G_0$, where G_2 is denoted by (N_2, T_2, S_2, P_2).
2. While there exists a pair of production rules of the form $X_1 \to a\ [R_1]$ and $X_2 \to a\ [R_2]$ in P_2 $(X_1 \neq X_2)$ do:
   (a) Let Y be a new non-terminal symbol and define a substitution $\sigma = [X_1/Y,\ X_2/Y]$.
   (b) Let $G_3 := (N_2 \cup \{Y\} - \{X_1, X_2\},\ T_2,\ \sigma(S_2),\ P_3)$, where $P_3 = \sigma(P_2 \cup \{Y \to a\ [R_1|R_2]\} - \{X_1 \to a\ [R_1], X_2 \to a\ [R_2]\})$.
   (c) Let $G_2 := G_3$, where G_2 is denoted by (N_2, T_2, S_2, P_2).
3. Return G_2. □

Example 4. Consider Example 1 where grammar G_0 is the input for the algorithm of Definition 8. In a first step, rules $Z \to \mathit{catalog}[(Y \mid L)^*]$ and $C \to \mathit{catalog}[B^*]$ are replaced by $C_Z \to \mathit{catalog}[(Y \mid L)^* \mid B^*]$. Following the same idea, rules $B \to \mathit{book}[A^+.T.D.W]$ and $Y \to \mathit{book}[A^+.T.D.P?.E]$ are replaced by $B_Y \to \mathit{book}[A^+.T.D.W \mid A^+.T.D.P?.E]$. This change implies changes on the right-hand side of other rules, consequently $C_Z \to \mathit{catalog}[(Y \mid L)^* \mid B^*]$ is changed into $C_Z \to \mathit{catalog}[(B_Y \mid L)^* \mid B_Y^*]$. In this way we obtain the grammar G_2 as shown in Example 1. □

It can be shown that our algorithm stops, generating a local tree grammar in normal form (see [CHMR09]). The main result of this section is stated below.

Theorem 1. *The grammar returned by the algorithm of Definition 8 generates the least local tree language that contains $L(G_0)$.* □

Example 5. Let the RTG G_0 be the input of the algorithm of Definition 8 and G be the resulting LTG:

G_0:	G:
$S \to a[A.A]$	$Y \to a[Y.Y \mid B]$
$A \to a[B]$	$B \to b[\epsilon]$
$B \to b[\epsilon]$	

Notice that $L(G_0)$ contains just the tree $t = a(a(b), a(b))$ and that $t \in L(G)$. □

3.2 Transforming a RTG into a STTG

Given a regular tree grammar G_0, we are interested in the definition of the *least* single-type tree language that contains the language generated by G_0. The new language will be described by a single-type grammar. The algorithm described below obtains a new grammar by transforming G_0. Roughly speaking, for each production rule $A \to a\,[R]$ in G_0, an equivalence relation is defined on the non-terminals of R, so that all competing non-terminals of R are in the same equivalence class. These equivalence classes form the non-terminals of the new grammar. Let $G_0 = (N_0, T_0, S_0, P_0)$ be a RTG in reduced normal form.

Definition 9 (Grouping competing non-terminals). Let $\parallel$ be the relation on N_0 defined by: for all $A, B \in N_0$, $A \parallel B$ iff $A = B$ or A and B are competing in P_0. For any $\chi \in \mathcal{P}(N_0)$, let $\parallel_\chi$ be the restriction of $\parallel$ to the set χ ($\parallel_\chi$ is defined only for elements of χ).

Lemma 2. *Since G_0 is in normal form, $\parallel_\chi$ is an equivalence relation for any $\chi \in \mathcal{P}(N_0)$.* □

Some notation used in this section:

– $N(R)$ denotes the set of non-terminals occurring in a regular expression R.
– For any $\chi \in \mathcal{P}(N_0)$ and any $A \in \chi$, $\hat{A}^\chi$ denotes the equivalence class of A w.r.t. relation $\parallel_\chi$. In other words, $\hat{A}^\chi$ contains A and the non-terminals of χ that are competing with A in P_0.
– $\sigma_{N(R)}$ is the substitution defined over $N(R)$ by $\forall A \in N(R)$, $\sigma_{N(R)}(A) = \hat{A}^{N(R)}$. By extension, $\sigma_{N(R)}(R)$ is the regular expression obtained from R by replacing each non-terminal A in R by $\sigma_{N(R)}(A)$.

Definition 10 (RTG into STTG Transformation). Let $G_0 = (N_0, T_0, S_0, P_0)$ be a regular tree grammar in reduced normal form. We define a new regular tree grammar $G = (N, T, S, P)$, obtained from G_0, according to the steps:

1. Let $G = (\mathcal{P}(N_0), T_0, S, P)$ where:
   - $S = \{\hat{A}^{S_0} \mid A \in S_0\}$,
   - $P = \{\, \{A_1, \ldots, A_n\} \to a\,[\sigma_{N(R)}(R)] \mid A_1 \to a[R_1], \ldots, A_n \to a[R_n] \in P_0,$
     $R = (R_1 | \cdots | R_n)\}$,
     where $\{A_1, \ldots, A_n\}$ denotes *each* possible set of competing non-terminals.
2. Remove all unreachable non-terminals and rules in G, then return it. □

Our generation of STTG from RTG is based on grouping competing non-terminals into equivalence classes. In the new grammar, each non-terminal is formed by a set of non-terminals of N_0. When competing non-terminals which appear in the same regular expression R in G_0 are identified, the sets that contain them form non-terminal symbols. The production rules having these new symbols as left-hand side are obtained from those rules containing the competing symbols in G_0. Although this amounts to an exponential number of non-terminal, we have notice that, in practice, this explosion is not common (notice that unreachable rules are removed at step 2). The algorithm version presented in Definition 10 eases our proofs. An optimized version, where just the needed non-terminals are generated, is given in Section 4.

Example 6. Consider a non-STTG grammar G_0 having the following set P_0 of productions rules (*Image* is the start symbol):

$Image \to image[Frame1 \mid Frame2 \mid Background.Foreground]$
$Frame1 \to frame[Frame1.Frame1 \mid \epsilon]$
$Frame2 \to frame[Frame2.Frame2.Frame2 \mid \epsilon]$
$Background \to back[Frame1]$ and $Foreground \to fore[Frame2]$.

Grammar G_0 defines different ways of decomposing an image: recursively into two or three frames or by describing the background and the foreground separately. Moreover, the background (resp. the foreground) is described by binary decompositions (resp. ternary decompositions). In other words, the language of G_0 contains the union of the trees: *image(bin(frame))*; *image(ter(frame))* and *image (back (bin (frame)), fore (ter (frame)))* where *bin* (resp. *ter*) denotes the set of all binary (resp. ternary) trees that contains only the symbol *frame*.

The algorithm returns G, which contains the rules below (the start symbol is $\{Image\}$):

$$\begin{aligned}
&\{Image\} \to image[\{Frame1, Frame2\} \mid \{Frame1, Frame2\} \\
&\qquad\qquad \mid \{Background\}.\{Foreground\}] \\
&\{Background\} \to back[\{Frame1\}] \\
&\{Foreground\} \to fore[\{Frame2\}] \\
&\{Frame1, Frame2\} \to frame[\epsilon \mid \{Frame1, Frame2\}.\{Frame1, Frame2\} \mid \epsilon \\
&\qquad\qquad \mid \{Frame1, Frame2\}.\{Frame1, Frame2\}.\{Frame1, Frame2\}] \\
&\{Frame1\} \to frame[\{Frame1\}.\{Frame1\} \mid \epsilon] \\
&\{Frame2\} \to frame[\{Frame2\}.\{Frame2\}.\{Frame2\} \mid \epsilon]
\end{aligned}$$

Note that some regular expressions could be simplified. G is a STTG that generates the union of *image(tree(frame))* and *image (back (bin (frame)), fore (ter (frame)))* where *tree* denotes the set of all trees that contain only the symbol *frame* and such that each node has 0 or 2 or 3 children. Let $L_G(X)$ denote the language obtained by deriving in G the non-terminal X. Actually, $L_G(\{Frame1, Frame2\})$ is the least STTL that contains $L_{G_0}(Frame1) \cup L_{G_0}(Frame2)$. □

Theorem 2. *The grammar returned by the algorithm of Definition 10 generates the least STTL that contains $L(G_0)$.* □

The rest of this sub-section is a proof sketch of the previous theorem. The notations are those of Definition 10. The proof somehow looks like the proof concerning the transformation of a RTG into a LTG (see [CHMR09] for details). However it is more complicated since in a STTL (and unlike what happens in a LTL), the confusion between $t|_p = a(w)$ and $t'|_{p'} = a(w')$ should be done only if position p in t has been generated by the same production rule as position p' in t', i.e. the symbols occurring in t and t' along the paths going from root to p (resp. p' in t') are the same. This is why, in Definition 11, we introduce notation $path(t, p)$ to denote these symbols. First, we enunciate some properties.

Lemma 3. *Let $\chi \in \mathcal{P}(N_0)$ and $A, B \in \chi$ $(A \neq B)$. Then $\hat{A}^\chi$ and $\hat{B}^\chi$ are not competing in P.* □

Example 7. Given the grammar of Example 6, let $\chi = \{Frame1, Frame2, Background\}$. The equivalence classes induced by $\|_\chi$ are $\widehat{Frame1}^\chi = \widehat{Frame2}^\chi = \{Frame1, Frame2\}$; $\widehat{Background}^\chi = \{Background\}$; which are non-competing non-terminals in P.

Lemma 4. *G is a STTG.* □

The next lemma establishes the basis for proving that the language generated by G contains the language generated by G_0. It considers the derivation process over G_0 at any step (supposing that this step is represented by a derivation tree t) and proves that, in this case, at the same derivation step over G, we can obtain a tree t' having all the following properties: (i) the set of positions is the same for both trees ($Pos(t) = Pos(t')$); (ii) positions associated to terminal are identical in both trees; (iii) if position p is associated to a non-terminal A in t then position $p \in Pos(t')$ is associated to the equivalence class $\hat{A}^{\chi}$ for some $\chi \in \mathcal{P}(N_0)$ such that $A \in \chi$.

Lemma 5. *Let $Y \in S_0$. If G_0 derives:*
$t_0 = Y \to \cdots \to t_{n-1} \to_{[p_n,\, A_n \to a_n[R_n]]} t_n$ *then G can derive:* $t'_0 = \hat{Y}^{S_0} \to \cdots \to t'_{n-1} \to_{[p_n,\, \hat{A_n}^{\chi_n} \to a_n[\sigma_{N(R_n|\cdots)}(R_n|\cdots)]]} t'_n$ *s.t.* $\forall i \in \{0, \ldots, n\}, Pos(t'_i) = Pos(t_i) \wedge$
$\forall p \in Pos(t_i)$*:* $(t_i(p) \in T_0 \implies t'_i(p) = t_i(p)) \wedge$
$$(t_i(p) = A \in N_0 \implies \exists \chi \in \mathcal{P}(N_0),\, A \in \chi \wedge t'_i(p) = \hat{A}^{\chi})$$

Proof. The proof is by induction in the length of the derivation process.
For $n = 0$, the property holds because $t_0(\epsilon) = Y$ and $t'_0(\epsilon) = \hat{Y}^{\chi}$ with $\chi = S_0$ and $Y \in \chi$. Induction step: Assume the property for $n - 1 \in \mathbb{N}$. By hypothesis $t_{n-1} \to_{[p_n,\, A_n \to a_n[R_n]]} t_n$, then $t_{n-1}(p_n) = A_n \in N_0$. By ind. hyp., $t'_{n-1}(p_n) = \hat{A_n}^{\chi_n}$ for some $\chi_n \in \mathcal{P}(N_0)$, and $A_n \in \chi_n$.
By construction of P, $\hat{A_n}^{\chi_n} \to a_n[\sigma_{N(R_n|\cdots)}\ (R_n|\cdots)] \in P$.
Thus $t'_{n-1} \to_{[p_n,\, \hat{A_n}^{\chi_n} \to a_n[\sigma_{N(R_n|\cdots)}(R_n|\cdots)]]} t'_n = t'_{n-1}[p_n \leftarrow a_n[\sigma_{N(R_n|\cdots)}(w)]]$ whereas $t_n = t_{n-1}[p_n \leftarrow a_n(w)]$ and $w \in L(R_n)$. Consequently $t'_n(p_n) = a_n = t_n(p_n)$ and $\forall i \in \mathbb{N}$, $t_n(p_n.i) = B \in N(R_n) \subseteq N(R_n|\cdots) \wedge t'_n(p_n.i) = \hat{B}^{N(R_n|\cdots)}$.

Example 8. Given the grammar of Example 6, consider trees t, t' and t'' in Figure 1 obtained after three steps in the derivation process: t is a derivation tree for G_0 while t' and t'' are for G. Tree t' is the one that corresponds to t according to Lemma 5. Notice that t'' is a tree that can also be derived from G, but it is not in $L(G_0)$ (indeed, since $Pos(t) \neq Pos(t'')$, tree t'' does not have the properties required in Lemma 5). □

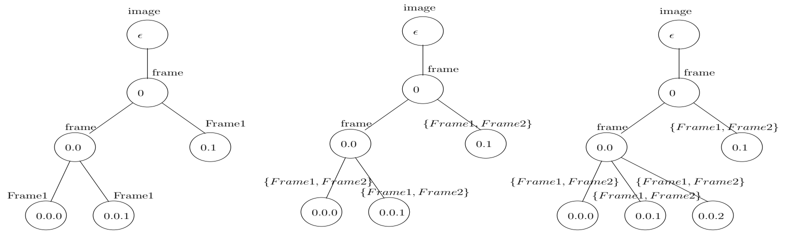


Fig. 1. Derivation trees t, t' and t''

The following corollary proves that the language of the new grammar G, proposed by Definition 10, contains the original language of G_0.

Corollary 2. *$L(G_0) \subseteq L(G)$.* □

In the rest of this section we work on proving that $L(G)$ is the least STTL that contains $L(G_0)$. To prove this property, we first need to prove some properties over STTLs. We start by considering paths in a tree. We are interested by paths starting on the root and achieving a given position p in a tree t. Paths are defined as a sequence of labels. For example, $path(a(b, c(d)), 1) = a.c$.

Definition 11 (Path in a tree t to a position p). Let t be a tree and $p \in Pos(t)$. We define $path(t, p)$ as being the word of symbols occurring in t along the branch going from the root to position p. Formally, $path(t, p)$ is recursively defined by : $path(t, \epsilon) = t(\epsilon)$ and $path(t, p.i) = path(t, p).t(p.i)$ where $i \in \mathbb{N}$. □

Given a STTG G, let us consider the derivation process of two trees t and t' belonging to $L(G)$. The following lemma proves that positions (p in t and p' in t') having identical paths are derived by using the same rules. A consequence of this lemma (when $t' = t$ and $p' = p$) is the well known result about the unicity in the way of deriving a given tree with a STTG [ML02].

Lemma 6. *Let G' be a STTG, let $t, t' \in L(G')$.*
Let $X \rightarrow^*_{[p_i, rule_{p_i}]} t$ be a derivation of t and $X' \rightarrow^*_{[p'_i, rule'_{p'_i}]} t'$ be a derivation of t' by G' (X, X' are start symbols). Then $\forall p \in Pos(t)$, $\forall p' \in Pos(t')$, $(path(t, p) = path(t', p') \implies rule_p = rule'_{p'})$

In a STTL, it is possible to exchange sub-trees that have the same paths.

Lemma 7. *(also in [MNSB06, Prop 6.3 and 6.5])*
Let G' be a STTG. $\forall t, t' \in L(G')$, $\forall p \in Pos(t)$, $\forall p' \in Pos(t')$, $(path(t, p) = path(t', p') \implies t'[p' \leftarrow t|_p] \in L(G'))$

Example 9. Let G be the grammar of Example 6. Consider a tree t as shown in Figure 2. Exchanging subtrees $t|_{0.0}$ and $t|_{0.1}$ gives us a new tree t''. Both t and t'' are in $L(G)$. □

The following lemma expresses what the algorithm of Definition 10 does. Given a forest $w = (t_1, \ldots, t_n)$, recall that $w(\epsilon) = \langle t_1(\epsilon), \ldots, t_n(\epsilon)\rangle$, i.e. $w(\epsilon)$ is the tuple of the top symbols of w.

Lemma 8. *$\forall t \in L(G), \forall p \in Pos(t)$, $t|_p = a(w) \implies \exists t' \in L(G_0), \exists p' \in pos(t'), t'|_{p'} = a(w') \wedge w'(\epsilon) = w(\epsilon) \wedge path(t', p') = path(t, p)$.* □

Example 10. Let G be the grammar of Example 6 and t the tree of Figure 2. Let $p = 0$. Using the notations of Lemma 8, $t|_0 = frame(w)$ where $w = \langle frame(frame, frame, frame),\ frame(frame, frame)\rangle$. We have $t \notin L(G_0)$. Let $t' = image(frame(frame(frame, frame),\ frame)) \in L(G_0)$ and (with $p' = p = 0$) $t'|_{p'} = frame(w')$ where $w' = \langle frame(frame, frame), frame\rangle$. Thus $w'(\epsilon) = w(\epsilon)$. Note that others $t' \in L(G_0)$ suit as well. □

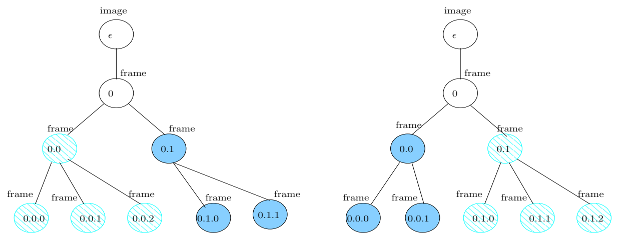


Fig. 2. Trees t and t'' with sub-tree exchange

We end this section by proving that the grammar obtained by our algorithm generates the least STTL which contains $L(G_0)$.

Lemma 9. *Let L' be a STTL s.t. $L(G_0) \subseteq L'$. Let $t \in L(G)$. Then, $\forall p \in Pos(t), \exists t' \in L', \exists p' \in pos(t')$, $(t'|_{p'} = t|_p \wedge path(t', p') = path(t, p))$.* □

Proof. We define the relation $\sqsupset$ over $Pos(t)$ by $p \sqsupset q \iff \exists i \in \mathbb{N}, p.i = q$. Since $Pos(t)$ is finite, $\sqsupset$ is noetherian. The proof is by noetherian induction on $\sqsupset$. Let $p \in pos(t)$. Let us write $t|_p = a(w)$. From Lemma 8, we know that:
$\exists t' \in L(G_0), \exists p' \in pos(t'), t'|_{p'} = a(w') \wedge w'(\epsilon) = w(\epsilon)\ \wedge path(t', p') = path(t, p)$.
Thus, $t|_p = a(a_1(w_1), \ldots, a_n(w_n))$ and $t'|_{p'} = a(a_1(w'_1), \ldots, a_n(w'_n))$. Now let $p \sqsupset p.1$. By induction hypothesis:
$\exists t'_1 \in L', \exists p'_1 \in pos(t'_1), t'_1|_{p'_1} = t|_{p.1} = a_1(w_1)\ \wedge path(t'_1, p'_1) = path(t, p.1)$.
Notice that $t'_1 \in L'$, $t' \in L(G_0) \subseteq L'$, and L' is a STTL. Moreover $path(t'_1, p'_1) = path(t, p.1) = path(t, p).a_1 = path(t', p').a_1 = path(t', p'.1)$.
As $path(t'_1, p'_1) = path(t', p'.1)$, from Lemma 7 applied on t'_1 and t', we get $t'[p'.1 \leftarrow t'_1|_{p'_1}] \in L'$. However $(t'[p'.1 \leftarrow t'_1|_{p'_1}])|_{p'} = a(a_1(w_1), a_2(w'_2), \ldots, a_n(w'_n))$ and
$path(t'[p'.1 \leftarrow t'_1|_{p'_1}], p') = path(t', p') = path(t, p)$.
By applying the same reasoning for positions $p.2, \ldots, p.n$, we get a tree $t'' \in L'$ s.t. $t''|_{p'} = t|_p$ and $path(t'', p') = path(t, p)$.

Corollary 3. *(when $p = \epsilon$, and then $p' = \epsilon$) Let L' be a STTL s.t. $L' \supseteq L(G_0)$. Then $L(G) \subseteq L'$.* □

4 Implementation

A prototype tool implementing our algorithms can be downloaded from [CHMR10]. It is developed using the ASF+SDF Meta-Environment [vdBHdJ$^+$01] and it is about 1000 lines of code.

The algorithm of Definitions 8 is implemented in a straightforward way. However, Definition 12 below gives an improved version of the algorithm of Definition 10. This new version avoids to generate unreachable non-terminals, and is suited for direct implementation. Indeed, it keeps a set of unprocessed non-terminals (denoted by U) which are accessible from the start symbol. We just compute those non-terminal symbols which are accessible from the start symbols of the grammar. More precisely, in Definition 12, we start by computing the equivalence classes of the start symbols of G_0 and we insert them to the set U, containing those non-terminals which are not yet processed. At each iteration of the while loop, an element of U is chosen as the new non-terminal for which a new production rule is going to be created. This non-terminal is added to the set N. At each step, the set U is updated by adding to it those non-terminals appearing on the right-hand side of the new production rule, filtering the non-terminals already processed.

Definition 12 (RTG into STTG Transformation). Let $G_0 = (N_0, T_0, S_0, P_0)$ be a (general) regular tree grammar. We define a new single-type tree grammar $G = (N, T, S, P)$, obtained from G_0, according to the following steps:

```
1. Let $S := \{\hat{A}^{S_0} \mid A \in S_0\}$; $G := (N := \emptyset, T := T_0, S, P := \emptyset)$; $U := S$;
2. While $U \neq \emptyset$ do:
   (a) Choose $\{A_1, \dots, A_n\} \in U$;
   (b) Let $U := U - \{\{A_1, \dots, A_n\}\}$; $N := N \cup \{\{A_1, \dots, A_n\}\}$;
   (c) Let $P := P \cup \{ \{A_1, \dots, A_n\} \to a\, \sigma_{N(R)}(R)$
                 $\mid A_1 \to a[R_1], \dots, A_n \to a[R_n] \in P_0,\ R = (R_1 | \cdots | R_n)\}$;
   (d) Let $U := (U \cup \{\hat{A}^{N(R)} | A \in N(R)\}) - N$;
   End While
3. Return $G$.
```
□

It is straightforward to see that the algorithm of Definition 12 generates the same STTG as that of Definition 10. In the worst case, the number of non-terminals of the STTG returned by both algorithms is exponential in the number of the non-terminals of the initial grammar G_0. However, in many examples, most non-terminals generated by step 1 of Definition 10 are unreachable, and thus are not generated by the implemented algorithm.

The example below represents a usual situation, in which the schemas of two different digital libraries (Grammars G_1 and G_2) are joined into one schema. *Library* and *Lib* are the start symbols. For lack of space, we do not depict the production rules $X \to x[\epsilon]$ such that $X \in$ {Author, Title, ISBN, Publisher, Date, ISSN, Editor, Year, Pages, Dimensions, Scale} and $x \in$ {author, title, isbn, publisher, date, issn, editor, year, pages, dim, scale} respectively. We apply the algorithm on $G_0 = G_1 \cup G_2$. Note that Author, Title, ISBN, Publisher, Date appear both in G_1 and G_2. It is not necessary to rename them before computing the union, since each of them generates only one terminal. G_0 contains 18 rules.

Productions rules of G_1	
Library	→ lib [Book*]
Book	→ book [Author.Title.ISBN.Publisher.Date]

Productions rules of G_2	
Lib	→ lib [(Mag | Record | Book2 | Map)*]
Mag	→ mag [Title.ISSN.Editor.Publisher.Date]
Record	→ rec [Title.Author.Date]
Book2	→ book [ISBN.Title.Author.Publisher.Year.Pages]
Map	→ map [Editor.Dimensions.Scale.Year]

The resulting grammar, after applying the algorithm, is[1]:

Resulting production rules	
Library	→ lib [(Mag | Record | Book | Map)* | Book*]
Book	→ book [ISBN.Title.Author.Publisher.Year.Pages | Author.Title.ISBN.Publisher.Date]
Mag	→ mag [Title.ISSN.Editor.Publisher.Date]
Record	→ rec [Title.Author.Date]
Map	→ map [Editor.Dimensions.Scale.Year]

Notice that the new grammar has only 16 rules (included those with empty regular expressions on their right-hand side). Only 16 new non-terminals were created by our algorithm, that is two less than those from the original grammar. This shows that in a typical situation, our algorithm runs in acceptable time, even if the worst case is exponential.

5 Related Work

As discussed in [Flo05], traditional tools require the data schema to be developed prior to the creation of the data. Unfortunately, in several modern applications the schema often changes as the information grows and different people have inherently different ways of modeling the same information. Complete elimination of the schema does not seem to be a solution since it assigns meaning to the data and thus helps automatic data search, comparison and processing. To find a balance, [Flo05] considers that we need to find how to automatically map schemas and vocabulary to each other and how to rewrite code written for a certain schema into code written for another schema describing the same domain.

Most existing work in the area of XML schema evolution concerns the second proposed solution. For instance, in [GMR05] the idea is to keep track of the updates made to the schema and to identify the portions of the schema that require validation. Our approach aims to be included in the first proposed solution of [Flo05] since it allows the conservative evolution of schemas. Indeed, our method extends the work in [BDH$^+$04, dHM07] which considers the conservative evolution of LTG by proposing the evolution of regular expressions. Contrary to this, the present paper proposes schema evolution in a global perspective, dealing with the tree grammars as a whole. We also consider the evolution of STTG.

Our proposal is inspired in some grammar inference methods (such as those in [BM03, BM06] which deal with ranked tree languages) that return a tree grammar or a tree automaton from a set of positive examples (see [Ang92, Sak97] for surveys). Our method deals with unranked trees, starts from a given RTG G_0 (representing a set of positive examples) and finds the least LTG or STTG that contains $L(G_0)$. As we consider an initial tree grammar we are not exactly inserted in the learning domain, but their methods inspire us and give us tools to solve our problem, namely, the evolution of a original schema (and not the extraction of a new schema).

[1] For more readability, the non-terminals have been renamed. *Library* is the start symbol. We have chosen not to simplify the regular expressions, showing them as produced by our algorithm. Our implementation do not implement any simplification yet.

Several papers (such as in [GGR$^+$00, Chi01, BNST06, BNV07]) deal with XML schema inference. In [BNST06] DTD inference consists in an inference of regular expressions from positive examples. As the seminal result from Gold [Gol67] shows that the class of all regular expressions cannot be learnt from positive examples, [BNST06] identifies classes of regular expressions that can be efficiently learnt. Their basic method consists in inferring a single occurrence automaton called SOA from a finite set of strings and to transform it to a SORE (regular expressions in which every element name can occur at most once). Their method is extended to deal with XMLSchema (XSD) in [BNV07].

In [AGM09], given a target global type of a distributed XML document, the authors propose a method to provide a subtype for each marked subtree such that (*i*) if each subtree verifies its subtype, the global type is verified and (*ii*) no extra restrictions than those imposed by the global type are introduced. Their approach consists in regarding the problem locally (each node and its children) and to find the FSA which should be associated to the children generated by external sources. Their approach can be seen as the inverse or ours. Let us suppose our library consortium example, with a big distributed XML document where nodes marked by functions are calls to different libraries. Their approach focus on defining the subtypes corresponding to each library supposing that a design is given. Our approach proposes to find the integration of different library subtypes by finding the least library type capable of verifying all library subtypes.

In [MNSB06, Th 10.3], it is shown that deciding whether a regular tree grammar has an equivalent LTG, or an equivalent STTG, is EXPTIME-complete. Using our algorithms, we can also solve these decision problems by computing the LTG (in linear time), or the STTG (in exponential time), that generates the least local (or single-type) language L containing the initial language L_0, and checking (in exponential time) that $L = L_0$.

6 Conclusion

This paper proposes algorithms that compute a *minimal* tree language, by finding the local or single-type grammar which generates it and which extends a given original regular grammar. The paper proves the correctness and the minimality of the generated grammars. A prototype has been implemented in order to show the feasibility of our approach. Our goal is to allow a given type to evolve encompassing the needs of the application using it. Indeed, we aim at developing tools for adapting XML message type of a web service to the needs of a composition.

It is encouraging to us to note that our work complements that of [MNSB06], where the complexity of deciding whether a regular tree grammar has an equivalent LTG or STTG is provided. In that work, the authors are interested in analyzing the actual expressive power of XSD. With some non-trivial amount of work, part of their theorem proofs can be used to produce an algorithm similar to ours. This emphasizes the relevance of our method whose usability is twofold: as a theoretical tool, it can help answering the decision problem announced in [MNSB06]; as an applied tool, it can easily be adapted to the context of digital libraries, web services, etc. Our work complements the proposals in [BDHM09, dHM07], since we consider not only DTD but also XSD, and adopts a global approach where all the tree grammar is taken into account as a whole.

Some aspects of our tool can be improved, in particular the conciseness of the regular expressions appearing in the generated grammars. We are working on improving and extending our approach to solve other questions related to type compatibility and evolution. Indeed, in this context, many other aspects may be taken into account such as integrity constraints (how they evolve when the schema evolves) and semantics of elements (how to deal with identical concepts named differently in each type). We intend not only to extend our work in these new directions but also to build an applied tool capable of comparing types or extracting some relevant parts of a type. Interesting theoretical problems are related to these applications.

References

[AGM09] Abiteboul, S., Gottlob, G., Manna, M.: Distributed XML design. In: PODS 2009: Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pp. 247–258. ACM, New York (2009)

[Ang92] Angluin, D.: Computational learning theory: survey and selected bibliography. In: STOC 1992: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing, pp. 351–369. ACM, New York (1992)

[BDHM09] Bouchou, B., Duarte, D., Halfeld Ferrari, M., Musicante, M.A.: Extending XML Types Using Updates. In: Hung (ed.) Services and Business Computing Solutions with XML: Applications for Quality Management and Best Processes, pp. 1–21. IGI Global (2009)

[BDH$^+$04] Bouchou, B., Duarte, D., Halfeld Ferrari, M., Laurent, D., Musicante, M.A.: Schema evolution for XML: A consistency-preserving approach. In: Fiala, J., Koubek, V., Kratochvíl, J. (eds.) MFCS 2004. LNCS, vol. 3153, pp. 876–888. Springer, Heidelberg (2004)

[BM03] Besombes, J., Marion, J.-Y.: Apprentissage des langages réguliers d'arbres et applications. Traitement automatique de langues 44(1), 121–153 (2003)

[BM06] Besombes, J., Marion, J.-Y.: Learning tree languages from positive examples and membership queries. Theoretical Computer Science (2006)

[BNST06] Bex, G.J., Neven, F., Schwentick, T., Tuyls, K.: Inference of concise DTDs from XML data. In: VLDB, pp. 115–126 (2006)

[BNV07] Bex, G.J., Neven, F., Vansummeren, S.: Inferring XML schema definitions from XML data. In: VLDB, pp. 998–1009 (2007)

[Chi01] Chidloviskii, B.: Schema extraction from XMLS data: A grammatical inference approach (2001)

[CHMR10] Chabin, J., Halfeld Ferrari, M., Musicante, M.A., Réty, P.: A software to transform a RTG into a LTG or a STTG (2010), `http://www.univ-orleans.fr/lifo/Members/rety/logiciels/RTGalgorithms.html`

[CHMR09] Chabin, J., Halfeld Ferrari, M., Musicante, M.A., Réty, P.: Minimal extensions of tree languages: Application to XML schema evolution. Technical Report RR-2009-06, LIFO (2009), `http://www.univ-orleans.fr/lifo/prodsci/rapports/RR/RR2009/RR-2009-06.pdf`

[dHM07] da Luz, R., Halfeld Ferrari, M., Musicante, M.A.: Regular expression transformations to extend regular languages (with application to a datalog XML schema validator). Journal of Algorithms (Special Issue) 62(3-4), 148–167 (2007)

[Flo05] Florescu, D.: Managing semi-structured data. ACM Queue 3(8), 18–24 (2005)

[GGR$^+$00] Garofalakis, M.N., Gionis, A., Rastogi, R., Seshadri, S., Shim, K.: Xtract: A system for extracting document type descriptors from XML documents. In: SIGMOD Conference, pp. 165–176 (2000)

[GMR05] Guerrini, G., Mesiti, M., Rossi, D.: Impact of XML schema evolution on valid documents. In: WIDM 2005: Proceedings of the 7th annual ACM international workshop on Web information and data management, pp. 39–44. ACM Press, New York (2005)

[Gol67] Gold, E.M.: Language identification in the limit. Information and Control 10(5), 447–474 (1967)

[ML02] Mani, M., Lee, D.: XML to Relational Conversion using Theory of Regular Tree Grammars. In: VLDB Workshop on EEXTT, pp. 81–103. Springer, Heidelberg (2002)

[MLMK05] Murata, M., Lee, D., Mani, M., Kawaguchi, K.: Taxonomy of XML schema languages using formal language theory. ACM Trans. Inter. Tech. 5(4), 660–704 (2005)

[MNSB06] Martens, W., Neven, F., Schwentick, T., Bex, G.J.: Expressiveness and complexity of XML schema. ACM Trans. Database Syst. 31(3), 770–813 (2006)

[PV00] Papakonstantinou, Y., Vianu, V.: DTD inference for views of XML data. In: PODS - Symposium on Principles of Database System, pp. 35–46. ACM Press, New York (2000)

[Sak97] Sakakibara, Y.: Recent advances of grammatical inference. Theor. Comput. Sci. 185(1), 15–45 (1997)

[vdBHdJ$^+$01] van den Brand, M., Heering, J., de Jong, H., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P., Scheerder, J., Vinju, J., Visser, E., Visser, J.: The ASF+SDF meta-environment: a component-based language development environment. Electronic Notes in Theoretical Computer Science 44(2) (2001)

Tracking Down the Origins of Ambiguity in Context-Free Grammars

H.J.S. Basten

Centrum Wiskunde & Informatica
P.O. Box 94079
NL-1090 GB Amsterdam, The Netherlands

Abstract. Context-free grammars are widely used but still hindered by ambiguity. This stresses the need for detailed detection methods that point out the sources of ambiguity in a grammar. In this paper we show how the approximative Noncanonical Unambiguity Test by Schmitz can be extended to conservatively identify production rules that do not contribute to the ambiguity of a grammar. We prove the correctness of our approach and consider its practical applicability.

1 Introduction

Context-free grammars (CFGs) are widely used in various fields, like for instance programming language development, natural language processing, or bioinformatics. They are suitable for the definition of a wide range of languages, but their possible ambiguity can hinder their use. Designed ambiguities are not uncommon, but accidentally introduced ambiguities are unwanted. Ambiguities are very hard to detect by hand, so automated ambiguity checkers are welcome tools.

Despite the fact the CFG ambiguity problem is undecidable in general [5,7,6], various detection schemes exist. They can roughly be divided into two categories: exhaustive methods and approximative ones. Methods in the first category exhaustively search the usually infinite set of derivations of a grammar, while the latter ones apply approximation to limit their search space. This enables them to always terminate, but at the expense of potentially incorrect reports. Exhaustive methods do produce precise reports, but only if they find ambiguity before they are halted, because they obviously cannot be run forever.

Because of the undecidability it is impossible to always terminate with a correct and detailed report. The challenge is to develop a method that gives the most precise answer in the time available. In this paper we propose to combine exhaustive and approximative methods as a step towards this goal. We show how to extend the Regular Unambiguity Test and Noncanonical Unambiguity Test [11] to improve the precision of their approximation and that of their ambiguity reports. The extension enables the detection of production rules that do not contribute to the ambiguity of a grammar. These are already helpful reports for the grammar developer, but can also be used to narrow the search space of other detection methods. In an earlier study [3] we witnessed significant reductions in the run-time of exhaustive methods due to our grammar filtering.

1.1 Related Work

The original Noncanonical Unambiguity Test by Schmitz is an approximative test for the unambiguity of a grammar. The approximation it applies is always conservative, so it can only find a grammar to be *unambiguous* or *potentially ambiguous*. Its answers always concern the grammar as a whole, but the reports of a prototype implementation [12] by the author also contain clues about the production rules involved in the potential ambiguity. However, these are very abstract and hard to understand. The extensions that we present do result in precise reports, while remaining conservative.

Another approximative ambiguity detection scheme is the "Ambiguity Checking with Language Approximation" framework [4] by Brabrand, Giegerich and Møller. The framework makes use of a characterization of ambiguity into horizontal and vertical ambiguity to test whether a certain production rule can derive ambiguous strings. The difference with our approach is that we test whether a production rule is vital for the existence of parse trees of ambiguous strings.

1.2 Overview

We start with background information about grammars and languages in Section 2. Then we repeat the definition of the Regular Unambiguity (RU) Test in Section 3. In Section 4 we explain how the RU Test can be extended to identify sets of parse trees of unambiguous strings. From these parse trees we can identify harmless production rules as explained in Section 5. Section 6 explains the Noncanonical Unambiguity (NU) Test, an improvement over the RU Test, and also shows how it improves the effect of our parse tree and production rule filtering. In Section 7 we describe how our approach can be used iteratively to increase its precision. Finally, Section 9 contains the conclusion.

We prove our results in an accompanying technical report [2].

2 Preliminaries

This section gives a quick overview of the theory of grammars and languages, and introduces the notational convention used throughout this document. For more background information we refer to [9,14].

2.1 Context-Free Grammars

A context-free grammar G is a 4-tuple (N, T, P, S) consisting of:

– N, a finite set of *nonterminals*,
– T, a finite set of *terminals* (the alphabet),
– P, a finite subset of $N \times (N \cup T)^*$, called the *production rules*,
– S, the *start symbol*, an element from N.

We use V to denote the set $N \cup T$, and V' for $V \cup \{\varepsilon\}$. The following characters are used to represent different symbols and strings: $a, b, c, \ldots$ represent terminals,

$A, B, C, \ldots$ represent nonterminals, X, Y, Z represent either nonterminals or terminals, $\alpha, \beta, \gamma, \ldots$ represent strings in V^*, $u, v, w, \ldots$ represent strings in T^*, ε represents the empty string.

A production (A, α) in P is written as $A \rightarrow \alpha$. We use the function $\mathsf{pid} : P \rightarrow \mathbb{N}$ to relate each production to a unique identifier. An *item* [10] indicates a position in the right hand side of a production using a dot. Items are written like $A \rightarrow \alpha \bullet \beta$.

The relation $\Longrightarrow$ denotes direct derivation, or derivation in one step. Given the string $\alpha B \gamma$ and a production rule $B \rightarrow \beta$, we can write $\alpha B \gamma \Longrightarrow \alpha \beta \gamma$ (read $\alpha B \gamma$ directly derives $\alpha \beta \gamma$). The symbol $\Longrightarrow^*$ means "derives in zero or more steps". A sequence of derivation steps is simply called a *derivation*. Strings in V^* are called *sentential forms*. We call the set of sentential forms that can be derived from S of a grammar G, the *sentential language* of G, denoted $\mathcal{S}(G)$. A sentential form in T^* is called a *sentence*. The set of all sentences that can be derived from S of a grammar G is called the *language* of G, denoted $\mathcal{L}(G)$.

We assume every nonterminal A is *reachable* from S, that is $\exists \alpha A \beta \in \mathcal{S}(G)$. We also assume every nonterminal is *productive*, meaning $\exists u : A \Longrightarrow^* u$.

The *parse tree* of a sentential form α describes how α is derived from S, but disregards the order of the derivation steps. To represent parse trees we use bracketed strings (See Section 2.3). A grammar G is ambiguous iff there is at least one string in $\mathcal{L}(G)$ for which multiple parse trees exist.

2.2 Bracketed Grammars

From a grammar $G = (N, T, P, S)$ a *bracketed grammar* G_b can be constructed, by adding unique terminals to the beginning and end of every production rule [8]. The bracketed grammar G_b is defined as the 4-tuple (N, T_b, P_b, S), where:

- $T_b = T \cup T_\langle \cup T_\rangle$,
- $T_\langle = \{ \langle_i \mid \exists p \in P : i = \mathsf{pid}(p) \}$,
- $T_\rangle = \{ \rangle_i \mid \exists p \in P : i = \mathsf{pid}(p) \}$,
- $P_b = \{ A \rightarrow \langle_i \alpha \rangle_i \mid A \rightarrow \alpha \in P, i = \mathsf{pid}(A \rightarrow \alpha) \}$.

V_b is defined as $T_b \cup N$, and V_b' as $V_b \cup \{\varepsilon\}$. We use $a_b, b_b, \ldots$ and X_b, Y_b, Z_b to represent symbols in respectively T_b and V_b. Similarly, $u_b, v_b, \ldots$ and $\alpha_b, \beta_b, \ldots$ represent strings in respectively T_b^* and V_b^*, The relation $\Longrightarrow_b$ denotes direct derivation using productions in P_b. The homomorphism h from V_b^* to V^* maps each string in $\mathcal{S}(G_b)$ to $\mathcal{S}(G)$. It is defined by $h(\langle_i) = \varepsilon$, $h(\rangle_i) = \varepsilon$, and $h(X) = X$.

2.3 Parse Trees

$\mathcal{L}(G_b)$ describes exactly all parse trees of all strings in $\mathcal{L}(G)$. $\mathcal{S}(G_b)$ describes exactly all parse trees of all strings in $\mathcal{S}(G)$. We divide it into two disjoint sets:

Definition 1. *The set of parse trees of ambiguous strings of G is $\mathcal{P}^a(G) = \{\alpha_b \mid \alpha_b \in \mathcal{S}(G_b), \exists \beta_b \in \mathcal{S}(G_b) : \alpha_b \neq \beta_b, h(\alpha_b) = h(\beta_b)\}$. The set of parse trees of unambiguous strings of G is $\mathcal{P}^u(G) = \mathcal{S}(G_b) \setminus \mathcal{P}^a(G)$.*

Example 1. Below is an example grammar (1) together with its bracketed version (2). The string aaa has two parse trees, $\langle_1\langle_2\langle_2\langle_3 a\rangle_3\langle_3 a\rangle_3\rangle_2\langle_3 a\rangle_3\rangle_2\rangle_1$ and $\langle_1\langle_2\langle_3 a\rangle_3\langle_2\langle_3 a\rangle_3\langle_3 a\rangle_3\rangle_2\rangle_2\rangle_1$, and is therefore ambiguous.

$$1 : S \to A, \qquad 2 : A \to AA, \qquad 3 : A \to a \tag{1}$$

$$1 : S \to \langle_1 A\rangle_1,\ 2 : A \to \langle_2 AA\rangle_2,\ 3 : A \to \langle_3 a\rangle_3 \tag{2}$$

We call the set of the smallest possible ambiguous sentential forms of G the *ambiguous core* of G. These are the ambiguous sentential forms that cannot be derived from other sentential forms that are already ambiguous. Their parse trees are the smallest indicators of the ambiguities in G.

Definition 2. *The set of parse trees of the* ambiguous core *of a grammar G is* $\mathcal{C}^a(G) = \{\alpha_b \mid \alpha_b \in \mathcal{P}^a(G),\ \neg\exists\beta_b \in \mathcal{P}^a(G) : \beta_b \Longrightarrow_b \alpha_b\}$

From $\mathcal{C}^a(G)$ we can obtain $\mathcal{P}^a(G)$ by adding all sentential forms reachable with $\Longrightarrow_b$. And since $\mathcal{C}^a(G) \subseteq \mathcal{P}^a(G)$ we get the following Lemma:

Lemma 1. *A grammar G is ambiguous iff $\mathcal{C}^a(G)$ is non-empty.*

Similar to $\mathcal{P}^u(G)$, we define the complement of $\mathcal{C}^a(G)$ as $\mathcal{C}^u(G) = \mathcal{S}(G_b)\backslash\mathcal{C}^a(G)$, for which holds that $\mathcal{P}^u(G) \subseteq \mathcal{C}^u(G)$.

Example 2. The two parse trees $\langle_1\langle_2\langle_2 AA\rangle_2 A\rangle_2\rangle_1$ and $\langle_1\langle_2 A\langle_2 AA\rangle_2\rangle_2\rangle_1$, of the ambiguous sentential form AAA, are in the ambiguous core of Grammar (1).

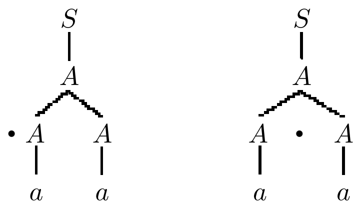
2.4 Positions

A *position* in a sentential form is an element in $V_b^* \times V_b^*$. The position (α_b, β_b) is written as $\alpha_b\bullet\beta_b$. $\mathsf{pos}(G_b)$ is the set of all positions in strings of $\mathcal{S}(G_b)$, defined as $\{\alpha_b\bullet\beta_b \mid \alpha_b\beta_b \in \mathcal{S}(G_b)\}$.

Every position in $\mathsf{pos}(G_b)$ is a position in a parse tree, and corresponds to an item of G. The item of a position can be identified by the closest enclosing $\langle_i$ and $\rangle_i$ pair around the dot, considering balancing. For positions with the dot at the beginning or the end we introduce two special items $\bullet S$ and $S\bullet$.

We use the function item to map a position to its item. It is defined by $\mathsf{item}(\gamma_b\bullet\delta_b) = A \to \alpha'\bullet\beta'$ iff $\gamma_b\bullet\delta_b = \eta_b\,\langle_i\,\alpha_b\bullet\beta_b\,\rangle_i\,\theta_b$, $A \to \langle_i\alpha'\beta'\rangle_i \in P_b$, $\alpha' \Longrightarrow_b^* \alpha_b$ and $\beta' \Longrightarrow_b^* \beta_b$, $\mathsf{item}(\bullet\alpha_b) = \bullet S$, and $\mathsf{item}(\alpha_b\bullet) = S\bullet$. Another function items returns the set of items used at all positions in a parse tree. It is defined as $\mathsf{items}(\alpha_b) = \{A \to \alpha\bullet\beta \mid \exists\gamma_b\bullet\delta_b : \gamma_b\delta_b = \alpha_b,\ A \to \alpha\bullet\beta = \mathsf{item}(\gamma_b\bullet\delta_b)\}$.

Example 3. The following shows the parse tree representations of the positions $\langle_1\langle_2\bullet\langle_3 a\rangle_3\langle_3 a\rangle_3\rangle_2\rangle_1$ and $\langle_1\langle_2\langle_3 a\rangle_3\bullet\langle_3 a\rangle_3\rangle_2\rangle_1$. We see that the first position is at item $A \to \bullet AA$ and the second is at $A \to A\bullet A$.



The function proditems maps a production rule to the set of all its items. It is defined as $\mathsf{proditems}(A \to \alpha) = \{A \to \beta\bullet\gamma \mid \beta\gamma = \alpha\}$. If a production rule is used to construct a parse tree, then all its items occur at one or more positions in the tree.

Lemma 2. $\forall \alpha_b \langle_i \beta_b \rangle_i \gamma_b \in \mathcal{S}(G_b) : \exists A \to \delta \in P$: $\mathsf{pid}(A \to \delta) = i$, $\mathsf{proditems}(A \to \delta) \subseteq \mathsf{items}(\alpha_b \langle_i \beta_b \rangle_i \gamma_b)$.

2.5 Automata

An *automaton* A is a 5-tuple (Q, Σ, R, Q_s, Q_f) where Q is the set of *states*, Σ is the input alphabet, R in $Q \times \Sigma \times Q$ is the set of *rules* or *transitions*, $Q_s \subseteq Q$ is the set of *start states*, and $Q_f \subseteq Q$ is the set of *final states*. A transition (q_0, a, q_1) is written as $q_0 \overset{a}{\longmapsto} q_1$. The language of an automaton is the set of strings read on all paths from a start state to an end state. Formally, $\mathcal{L}(A) = \{\alpha \mid \exists q_s \in Q_s, q_f \in Q_f : q_s \overset{\alpha}{\longmapsto}{}^* q_f\}$.

3 Regular Unambiguity Test

This section introduces the Regular Unambiguity (RU) Test [11] by Schmitz. The RU Test is an approximative test for the existence of two parse trees for the same string, allowing only false positives.

3.1 Position Automaton

The basis of the Regular Unambiguity Test is a *position automaton*, which describes all strings in $\mathcal{S}(G_b)$. The states of this automaton are the positions in $\mathsf{pos}(G_b)$. The transitions are labeled with elements from V_b.

Definition 3. *The position automaton*[1] $\Gamma(G)$ *of a grammar* G *is the tuple* (Q, V_b, R, Q_s, Q_f), *where*

- $Q = \mathsf{pos}(G_b)$,
- $R = \{\alpha_b\bullet X_b\beta_b \overset{X_b}{\longmapsto} \alpha_b X_b\bullet\beta_b \mid \alpha_b X_b \beta_b \in \mathcal{S}(G_b)\}$,
- $Q_s = \{\bullet\alpha_b \mid \alpha_b \in \mathcal{S}(G_b)\}$,
- $Q_f = \{\alpha_b\bullet \mid \alpha_b \in \mathcal{S}(G_b)\}$.

There are three types of transitions: *derives* with labels in $T_\langle$, *reduces* with labels in $T_\rangle$, and *shifts* of terminals and nonterminals in V. The symbols read on a path through $\Gamma(G)$ describe a parse tree of G. Thus, $\mathcal{L}(\Gamma(G)) = \mathcal{S}(G_b)$.

$\Gamma(G)$ contains a unique subgraph for each string in $\mathcal{S}(G_b)$. The string read by a subgraph can be identified by the positions on the nodes of the subgraph. Every position dictates the prefix read up until its node, and the postfix required to reach the end state of its subgraph. Therefore, every path that corresponds to a string in $\mathcal{L}(\Gamma(G))$ must pass all positions of that string.

[1] We modified the original definition of the position automaton to be able to explain our extensions more clearly. This does not essentially change the RU Test and NU Test however, since their only requirement on $\Gamma(G)$ is that it defines $\mathcal{S}(G_b)$.

Lemma 3. $\forall \alpha_b, \beta_b : \alpha_b \bullet \beta_b \in Q \Leftrightarrow \alpha_b\beta_b \in \mathcal{L}(\Gamma(G))$.

A grammar G is ambiguous iff two paths exist through $\Gamma(G)$ that describe different parse trees in $\mathcal{P}^a(G)$ — strings in $\mathcal{S}(G_b)$ — of the same string in $\mathcal{S}(G)$. We call such two paths an *ambiguous path pair.*

Example 4. The following shows the first part of the position automaton of the grammar from Example 1. It shows paths for parse trees S, $\langle_1 A\rangle_1$ and $\langle_1\langle_3 a\rangle_3\rangle_1$.

$$\begin{tikzcd}
{} \arrow[d] & \\
\boxed{\bullet S} \arrow[r, "S"] & \boxed{S\bullet}
\end{tikzcd}$$

$$\begin{tikzcd}
{} \arrow[d] & & & \\
\boxed{\bullet\langle_1 A\rangle_1} \arrow[r, "\langle_1"] & \boxed{\langle_1\bullet A\rangle_1} \arrow[r, "A"] & \boxed{\langle_1 A\bullet\rangle_1} \arrow[r, "\rangle_1"] & \boxed{\langle_1 A\rangle_1\bullet}
\end{tikzcd}$$

$$\begin{tikzcd}
{} \arrow[d] & & & & & \\
\boxed{\bullet\langle_1\langle_3 a\rangle_3\rangle_1} \arrow[r, "\langle_1"] & \boxed{\langle_1\bullet\langle_3 a\rangle_3\rangle_1} \arrow[r, "\langle_3"] & \boxed{\langle_1\langle_3\bullet a\rangle_3\rangle_1} \arrow[r, "a"] & \boxed{\langle_1\langle_3 a\bullet\rangle_3\rangle_1} \arrow[r, "\rangle_3"] & \boxed{\langle_1\langle_3 a\rangle_3\bullet\rangle_1} \arrow[r, "\rangle_1"] & \boxed{\langle_1\langle_3 a\rangle_3\rangle_1\bullet}
\end{tikzcd}$$

3.2 Approximated Position Automaton

If G has an infinite number of parse trees, the position automaton is also of infinite size. Checking it for ambiguous path pairs would take forever. Therefore the position automaton is approximated using equivalence relations on the positions. The approximated position automaton has equivalence classes of positions for its states. For every transition between two positions in the original automaton a new transition with the same label then exists between the equivalence classes that the positions are in. If an equivalence relation is used that yields a finite set of equivalence classes, the approximated automaton can be checked for ambiguous path pairs in finite time.

Definition 4. *Given an equivalence relation $\equiv$ on positions, the approximated position automaton $\Gamma_{\equiv}(G)$ of the automaton $\Gamma(G) = (Q, V_b, R, Q_s, Q_f)$, is the tuple $(Q_{\equiv}, V_b', R_{\equiv}, \{q_s\}, \{q_f\})$ where*

– *$Q_{\equiv} = Q/{\equiv} \cup \{q_s, q_f\}$, where $Q/{\equiv}$ is the set of non-empty equivalence classes over $\mathsf{pos}(G_b)$ modulo $\equiv$, defined as $\{[\alpha_b \bullet \beta_b]_{\equiv} \mid \alpha_b \bullet \beta_b \in Q\}$,*
– *$R_{\equiv} = \{[q_0]_{\equiv} \overset{X_b}{\longmapsto} [q_1]_{\equiv} \mid q_0 \overset{X_b}{\longmapsto} q_1 \in R\} \cup \{q_s \overset{\varepsilon}{\longmapsto} [q]_{\equiv} \mid q \in Q_s\} \cup \{[q]_{\equiv} \overset{\varepsilon}{\longmapsto} q_f \mid q \in Q_f\}$,*
– *q_s and q_f are respectively the start and final state.*

The paths through $\Gamma_{\equiv}(G)$ describe an overapproximation of the set of parse trees of G, thus $\mathcal{L}(\Gamma(G)) \subseteq \mathcal{L}(\Gamma_{\equiv}(G))$. So if no ambiguous path pair exists in $\Gamma_{\equiv}(G)$, grammar G is unambiguous. But if there is an ambiguous path pair, it is unknown if its paths describe real parse trees of G or approximated ones. In this case we say G is *potentially ambiguous.*

The item_0 Equivalence Relation. Checking for ambiguous paths in finite time also requires an equivalence relation with which $\Gamma_{\equiv}(G)$ can be built in finite

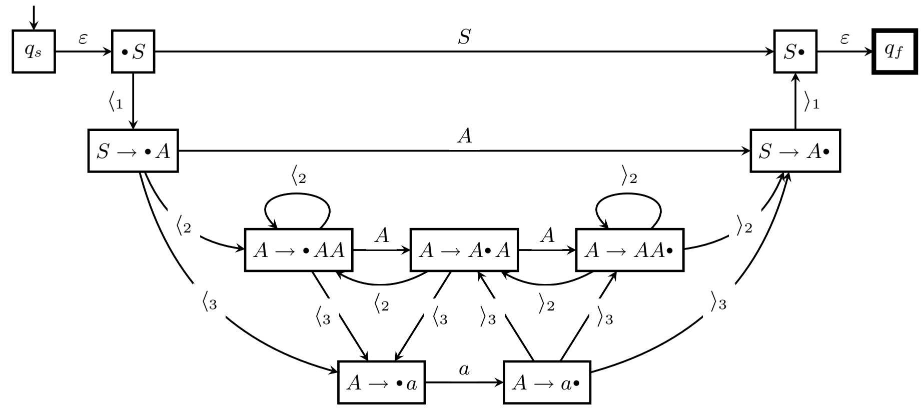


Fig. 1. The item_0 position automaton of the grammar of Example 1

time. A relation like that should enable the construction of the equivalence classes without enumerating all positions in $\mathsf{pos}(G_b)$. A simple but useful equivalence relation with this property is the item_0 relation [11]. Two positions are equal modulo item_0 if they are both at the same item.

Definition 5. $\alpha_b\bullet\beta_b \; \mathsf{item}_0 \; \gamma_b\bullet\delta_b$ *iff* $\mathsf{item}(\alpha_b\bullet\beta_b) = \mathsf{item}(\gamma_b\bullet\delta_b)$.

Intuitively the item_0 position automaton $\Gamma_{\mathsf{item}_0}(G)$ of a grammar resembles that grammar's LR(0) parse automaton [10]. The nodes are the LR(0) items of the grammar and the X and $\rangle$ edges correspond to the shift and reduce actions in the LR(0) automaton. The $\langle$ edges do not have LR(0) counterparts. Every item with the dot at the beginning of a production of S is a start node, and every item with the dot at the end of a production of S is an end node.

The difference between an LR(0) automaton and an item_0 position automaton is in the reductions. $\Gamma_{\mathsf{item}_0}(G)$ has reduction edges to every item that has the dot after the reduced nonterminal, while an LR(0) automaton jumps to a different state depending on the symbol that is at the top of the parse stack. As a result, a certain path through $\Gamma_{\mathsf{item}_0}(G)$ with a $\langle_i$ transition from $A \to \alpha\bullet B\gamma$ does not necessarily need to have a matching $\rangle_i$ transition to $A \to \alpha B\bullet\gamma$.

Example 5. Figure 1 shows the item_0 position automaton of the grammar of Example 1. Strings $\langle_1\langle_2\langle_3 a\rangle_3\rangle_1$ and $\langle_1\langle_3 a\rangle_3\rangle_1$ form an ambiguous path pair.

The item_0 relation can be combined with the look_k relation to get position automata that resemble LR(k) automata. This results in the item_k relation, which groups positions if they are equal modulo both item_0 and look_k. Two positions are equal modulo look_k if their first k terminal symbols after the dot are identical.

Definition 6. $\alpha_b\bullet\beta_b \; \mathsf{look}_k \; \gamma_b\bullet\delta_b$ *iff* $(\exists u, v, w : h(\beta_b) = uv,\ h(\delta_b) = uw,\ |u| = k)$ $\vee\ (h(\beta_b) = h(\delta_b) \wedge |h(\beta_b)| < k)$.

The RU Test becomes more precise with increasing k values, because then $\Gamma_{\mathsf{item}_k}(G)$ better approximates $\mathcal{S}(G)$.

3.3 Position Pair Automaton

The existence of ambiguous path pairs in a position automaton can be checked with a *position pair automaton*, in which every state is a pair of states from the position automaton. Transitions between pairs are described using the *mutual accessibility relation* ma.

Definition 7. *The* regular position pair automaton $\Pi^R_{\equiv}(G)$ *of* $\Gamma_{\equiv}(G)$ *is the tuple* $(Q^2_{\equiv}, V'^2_b, \mathsf{ma}, q^2_s, q^2_f)$*, where* ma *over* $Q^2_{\equiv} \times V'^2_b \times Q^2_{\equiv}$*, denoted by* $\rightrightarrows$ *, is the union of the following subrelations:*

$$\begin{aligned}
\mathsf{maDl} &= \{(q_0, q_1) \xrightarrow{(\langle_i,\varepsilon)} (q_2, q_1) \mid q_0 \overset{\langle_i}{\longmapsto} q_2\},\\
\mathsf{maDr} &= \{(q_0, q_1) \xrightarrow{(\varepsilon,\langle_i)} (q_0, q_3) \mid q_1 \overset{\langle_i}{\longmapsto} q_3\},\\
\mathsf{maS} &= \{(q_0, q_1) \xrightarrow{(X,X)} (q_2, q_3) \mid q_0 \overset{X}{\longmapsto} q_2 \wedge q_1 \overset{X}{\longmapsto} q_3,\ X \in V'\},\\
\mathsf{maRl} &= \{(q_0, q_1) \xrightarrow{(\rangle_i,\varepsilon)} (q_2, q_1) \mid q_0 \overset{\rangle_i}{\longmapsto} q_2\},\\
\mathsf{maRr} &= \{(q_0, q_1) \xrightarrow{(\varepsilon,\rangle_i)} (q_0, q_3) \mid q_1 \overset{\rangle_i}{\longmapsto} q_3\}.
\end{aligned}$$

Every path through this automaton from q^2_s to q^2_f describes two paths through $\Gamma_{\equiv}(G)$ that shift the same symbols. The language of $\Pi^R_{\equiv}(G)$ is thus a set of pairs of strings. A path indicates an ambiguous path pair if its two bracketed strings are different, but equal under the homomorphism h. Because $\mathcal{L}(\Gamma_{\equiv}(G))$ is an over-approximation of $\mathcal{S}(G_b)$, $\mathcal{L}(\Pi^R_{\equiv}(G))$ contains at least all ambiguous path pairs through $\Gamma(G)$.

Lemma 4. $\forall \alpha_b, \beta_b \in \mathcal{P}^a(G) : \alpha_b \neq \beta_b \wedge h(\alpha_b) = h(\beta_b) \Rightarrow (\alpha_b, \beta_b) \in \mathcal{L}(\Pi^R_{\equiv}(G))$.

4 Finding Parse Trees of Unambiguous Strings

The Regular Unambiguity Test described in the previous section can conservatively detect the unambiguity of a given grammar. If it finds no ambiguity we are done, but if it finds potential ambiguity this report is not detailed enough to be useful. In this section we show how the RU Test can be extended to identify parse trees of unambiguous strings. These will form the basis of more detailed ambiguity reports, as we will see in Section 5.

4.1 Unused Positions

From the states of $\Gamma_{\equiv}(G)$ that are not used on ambiguous path pairs, we can identify parse trees of unambiguous strings. For this we use the fact that every bracketed string that represents a parse tree of G must pass all its positions on its path through $\Gamma(G)$ (Lemma 3). Therefore, all positions in states of $\Gamma_{\equiv}(G)$ that are not used by any ambiguous path pair through $\Pi^R_{\equiv}(G)$ are positions in parse trees of unambiguous strings.

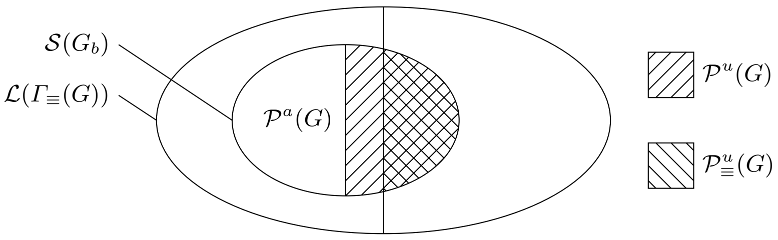


Fig. 2. Venn diagram showing the relationship between $\mathcal{S}(G_b)$ and $\mathcal{L}(\Gamma_\equiv(G))$. The vertical lines divide both sets in two: their parse trees of ambiguous strings (left) and parse trees of unambiguous strings (right).

Definition 8. *The set of states of $\Gamma_\equiv(G)$ that are used on ambiguous path pairs through $\Pi^R_\equiv(G)$ is $Q^a_\equiv =$*

$$\{q_0, q_1 \mid \exists \alpha_b, \beta_b, \alpha'_b, \beta'_b : \alpha_b\beta_b \neq \alpha'_b\beta'_b,\ q^2_s \xrightarrow{(\alpha_b,\alpha'_b)}{}^* (q_0, q_1) \xrightarrow{(\beta_b,\beta'_b)}{}^* q^2_f\}.$$

The set of states not used on ambiguous path pairs is $Q^u_\equiv = Q_\equiv \setminus Q^a_\equiv$.

Definition 9. *The set of parse trees of unambiguous strings of G that are identifiable with $\equiv$, is $\mathcal{P}^u_\equiv(G) = \{\alpha_b\beta_b \mid \exists q \in Q^u_\equiv : \alpha_b \bullet \beta_b \in q\}$.*

This set is always a subset of $\mathcal{P}^u(G)$, as illustrated by Fig. 2.

Theorem 1. *For all equivalence relations $\equiv$, $\mathcal{P}^u_\equiv(G) \subseteq \mathcal{P}^u(G)$.*

The positions in the states in $Q^a_\equiv$ and $Q^u_\equiv$ thus identify parse trees of respectively potentially ambiguous strings and certainly unambiguous strings. However, iterating over all positions in $\mathsf{pos}(G)$ is infeasible if this set is infinite. The used equivalence relation should therefore allow the direct identification of parse trees from the states of $\Gamma_\equiv(G)$.

For instance, a state in $\Gamma_{\mathsf{item}_0}(G)$ represents all parse trees in which a particular item appears. With this information we can identify production rules that only appear in parse trees in $\mathcal{P}^u_\equiv(G)$, as we will show in the next section.

4.2 Join Points

Gathering $Q^a_\equiv$ is also impossible in practice because it requires the inspection of all paths through $\Gamma_\equiv(G)$, of which there can be infinitely many. We therefore need a definition that can be calculated in finite time. For this we use the notion of *join points*. These are the points in $\Pi^R_\equiv(G)$ where we see that two different paths through $\Gamma_\equiv(G)$ potentially come together in the same state.

Definition 10. *The set of join points J in $\Pi^R_\equiv(G)$, over $Q^2_\equiv \times Q^2_\equiv$, is defined as $J = \{((q_0, q_1), (q_2, q_2)) \mid (q_0, q_1) \xrightarrow{(X_b, X'_b)} (q_2, q_2),\ q_0 \neq q_1,\ X_b \in T_\rangle \vee X'_b \in T_\rangle\}$.*

With J we then define the following alternative to $Q^a_\equiv$:

Definition 11. *The set of states in $\Gamma_{\equiv}(G)$ that are used in pairs of $\Pi^R_{\equiv}(G)$ that can reach, or can be reached by, a join point, is $Q^{a\prime}_{\equiv} =$*
$\{q_0, q_1 \mid \exists (p_0, p_1) \in J : q^2_s \rightrightarrows^* (q_0, q_1) \rightrightarrows^* p_0 \vee p_1 \rightrightarrows^* (q_0, q_1) \rightrightarrows^* q^2_f\}$.

This is a safe over-approximation of $Q^a_{\equiv}$, because all ambiguous path pairs through $\Gamma_{\equiv}(G)$ will eventually join in a certain state. It can be calculated by iterating over the edges of $\Pi^R_{\equiv}(G)$ to collect J, and then computing the images of the join points through ma^* and $(\mathsf{ma}^{-1})^*$. Both steps are linear in the number of edges in $\Pi^R_{\equiv}(G)$ (see [14] Chapter 2), which is worst case $\mathcal{O}(|Q_{\equiv}|^4)$.

5 Harmless Production Rules

In this section we show how we can use $Q^a_{\equiv}$ to identify production rules that do not contribute to the ambiguity of G. These are the production rules that can never occur in parse trees of ambiguous strings. We call them *harmless production rules.*

5.1 Finding Harmless Production Rules

A production rule is certainly harmless if it is only used in parse trees in $\mathcal{P}^u_{\equiv}(G)$. We should therefore search for productions that are never used on ambiguous path pairs of $\Pi^R_{\equiv}(G)$ that describe valid parse trees in G. We can find them by looking at the items of the positions in the states of $Q^a_{\equiv}$. If not all items of a production rule are used then the rule cannot be used in a valid string in $\mathcal{P}^a(G)$ (Lemma 2), and we know it is harmless.

Definition 12. *The set of items used on the ambiguous path pairs through $\Pi^R_{\equiv}(G)$ is $I^a_{\equiv} = \{A \to \alpha\bullet\beta \mid \exists q \in Q^a_{\equiv} : \exists \gamma_b\bullet\delta_b \in q : A \to \alpha\bullet\beta = \mathsf{item}(\gamma_b\bullet\delta_b)\}$.*

With it we can identify production rules of which all items are used:

Definition 13. *The set of potentially harmful production rules of G, identifiable from $\Pi^R_{\equiv}(G)$, is $P_{\text{hf}} = \{A \to \alpha \mid \mathsf{proditems}(A \to \alpha) \subseteq I^a_{\equiv}\}$.*

Because of the approximation it is uncertain whether or not they can really be used to form valid parse trees of ambiguous strings. Nevertheless, all the other productions in P will certainly not appear in parse trees of ambiguous strings.

Definition 14. *The set of harmless production rules of G, identifiable from $\Pi^R_{\equiv}(G)$, is $P_{\text{hl}} = P \setminus P_{\text{hf}}$.*

Theorem 2. *$\forall p \in P_{\text{hl}} : \neg\exists \alpha_b\langle_i\beta_b\rangle_i\gamma_b \in \mathcal{P}^a(G) : i = \mathsf{pid}(p)$.*

Example 6 in Section 7 shows finding P_{hl} for a small grammar.

5.2 Complexity

Finding P_{hf} comes down to building $\Pi^R_{\equiv}(G)$, finding $Q^{a\prime}_{\equiv}$, and enumerating all positions in all classes in $Q^{a\prime}_{\equiv}$ to find $I^{a}_{\equiv}$. The number of these classes is finite, but the number of positions might not be. It would therefore be convenient if the definition of the chosen equivalence relation could be used to collect $I^{a}_{\equiv}$ in finitely many steps. With the item_0 relation this is possible, because all the positions in a class are all in the same item.

Constructing $\Pi^R_{\mathsf{item}_0}(G)$ can be done in $\mathcal{O}(|G|^2)$ (see [11]), where $|G|$ is the number of items of G. After that, $Q^{a\prime}_{\mathsf{item}_0}$ can be gathered in $\mathcal{O}(|G|^4)$, because $|Q_{\mathsf{item}_0}|$ is linear with $|G|$. Since this is the most expensive step, the worst case complexity of finding P_{hf} with item_0 is therefore also $\mathcal{O}(|G|^4)$.

5.3 Grammar Reconstruction

Finding P_{hl} can be very helpful information for the grammar developer. Also, P_{hf} represents a smaller grammar that can be checked again more easily to find the true origins of ambiguity. However, the reachability and productivity properties of this smaller grammar might be violated because of the removed productions in P_{hl}. To restore these properties we have to introduce new terminals and productions, and a new start symbol. We must prevent introducing new ambiguities in this process.

From P_{hf} we can create a new grammar G' by constructing:

1. The set of defined nonterminals of P_{hf}: $N_{\mathrm{def}} = \{A \mid A \to \alpha \in P_{\mathrm{hf}}\}$.
2. The used but undefined nonterminals of P_{hf}:
   $N_{\mathrm{undef}} = \{B \mid A \to \alpha B \beta \in P_{\mathrm{hf}}\} \backslash N_{\mathrm{def}}$.
3. The unproductive nonterminals:
   $N_{\mathrm{unpr}} = \{A \mid A \in N_{\mathrm{def}}, \neg\exists u : A \Longrightarrow^* u$ using only productions in $P_{\mathrm{hf}}\}$.
4. The start symbols of P_{hf}: $S_{\mathrm{hf}} = \{A \mid A \in N_{\mathrm{def}}, \neg\exists B \to \beta A \gamma \in P_{\mathrm{hf}}\}$.
5. New terminal symbols t_A, b_A, e_A for each nonterminal A.
6. New productions to define a new start-symbol S':
   $P_{S'} = \{S' \to b_A A e_A \mid A \in S_{\mathrm{hf}}\}$.
7. Productions to complete the unproductive and undefined nonterminals:
   $P' = P_{\mathrm{hf}} \cup P_{S'} \cup \{A \to t_A \mid A \in N_{\mathrm{undef}} \cup N_{\mathrm{unpr}}\}$.
8. The new set of terminal symbols: $T' = \{a \mid A \to \beta a \gamma \in P'\}$.
9. Finally, the new grammar: $G' = (N_{\mathrm{def}} \cup N_{\mathrm{undef}} \cup \{S'\}, T', P', S')$.

Surrounding the nonterminals in S_{hf} with unique terminals at step 6 prevents the new rules of S' from being ambiguous with each other. The unique terminals at step 7 make sure we do not create new parse trees for existing strings in $\mathcal{L}(G)$.

6 Noncanonical Unambiguity Test

In this section we explain the Noncanonical Unambiguity (NU) Test [11], which is more precise than the Regular Unambiguity Test. It enables the identification of a larger set of irrelevant parse trees, namely the ones in $\mathcal{C}^u(G)$. From these we can also identify a larger set of harmless production rules and tree patterns.

6.1 Improving the Regular Unambiguity Test

The regular position pair automaton described in Section 3 checks all pairs of paths through a position automaton for ambiguity. However, it also checks some spurious paths that are unnecessary for identifying the ambiguity of a grammar.

These are the path pairs that derive the same unambiguous substring for a certain nonterminal. We can ignore these paths because in this situation there are also two paths in which the nonterminal was shifted instead of derived. For instance, consider paths $\langle_1\langle_2\langle_3 a\rangle_3\alpha_b\rangle_2\rangle_1$ and $\langle_1\langle_2\langle_3 a\rangle_3\beta_b\rangle_2\rangle_1$. If they form a pair in $\mathcal{L}(\Pi^R_{\equiv}(G))$ then the shorter paths $\langle_1\langle_2 A\alpha_b\rangle_2\rangle_1$ and $\langle_1\langle_2 A\beta_b\rangle_2\rangle_1$ will too (considering $A \rightarrow \langle_3 a\rangle_3 \in P_b$). In addition, if the first two paths form an ambiguous path pair, then these latter two will also, because $\langle_3 a\rangle_3$ does not contribute to the ambiguity. In this case we prefer the latter paths because they describe smaller parse trees than the first paths.

6.2 Noncanonical Position Pair Automaton

To avoid common unambiguous substrings we should only allow path pairs to take identical reduce transitions if they do not share the same substring since their last derives. To keep track of this property we add two extra boolean flags c_0 and c_1 to the position pairs. These flags tell for each position in a pair whether or not its path has been in conflict with the other, meaning it has taken different reduce steps as the other path since its last derive. A value of 0 means this has not occurred yet, and we are thus allowed to ignore an identical reduce transition.

All start pairs have both flags set to 0, and every derive step resets the flag of a path to 0. The flag is set to 1 if a path takes a *conflicting* reduce step, which occurs if the other path does not follow this reduce at the same time (for instance $\rangle_2$ in the parse trees $\langle_1\langle_2\langle_3 a\rangle_3\rangle_2\rangle_1$ and $\langle_1\langle_2\langle_3 a\rangle_3\rangle_1$). We use the predicate confl (called eff by Schmitz) to identify a situation like that.

$$\mathsf{confl}(q,i) = \exists u \in T^*_{\langle} : q \overset{u}{\longmapsto}{}^{*} q_f \vee (\exists q' \in Q_{\equiv}, X \in V \cup T_{\rangle} : X \neq \rangle_i, q \overset{uX}{\longmapsto}{}^{+} q') \quad (3)$$

It tells whether there is another shift or reduce transition other than $\rangle_i$ possible from q, ignoring $\langle$ steps, or if q is at the end of the automaton.

Definition 15. *The* noncanonical *position pair automaton $\Pi^N_{\equiv}(G)$ of $\Gamma_{\equiv}(G)$ is the tuple $(Q^p, V_b'^2, \mathsf{nma}, (q_s,0)^2, (q_f,1)^2)$, where $Q^p = (Q_{\equiv} \times \mathbb{B})^2$, and* nma *over $Q^p \times V_b'^2 \times Q^p$ is the* noncanonical *mutual accessibility relation, defined as the union of the following subrelations:*

$$\begin{aligned}
\mathsf{nmaDl} &= \{(q_0,q_1)c_0,c_1 \xrightarrow{(\langle_i,\varepsilon)} (q_2,q_1)0,c_1 \mid q_0 \overset{\langle_i}{\longmapsto} q_2\},\\
\mathsf{nmaDr} &= \{(q_0,q_1)c_0,c_1 \xrightarrow{(\varepsilon,\langle_i)} (q_0,q_3)c_0,0 \mid q_1 \overset{\langle_i}{\longmapsto} q_3\},\\
\mathsf{nmaS} &= \{(q_0,q_1)c_0,c_1 \xrightarrow{(X,X)} (q_2,q_3)c_0,c_1 \mid q_0 \overset{X}{\longmapsto} q_2,\ q_1 \overset{X}{\longmapsto} q_3,\ X \in V'\},\\
\mathsf{nmaCl} &= \{(q_0,q_1)c_0,c_1 \xrightarrow{(\rangle_i,\varepsilon)} (q_2,q_1)1,c_1 \mid q_0 \overset{\rangle_i}{\longmapsto} q_2,\ \mathsf{confl}(q_1,i)\},\\
\mathsf{nmaCr} &= \{(q_0,q_1)c_0,c_1 \xrightarrow{(\varepsilon,\rangle_i)} (q_0,q_3)c_0,1 \mid q_1 \overset{\rangle_i}{\longmapsto} q_3,\ \mathsf{confl}(q_0,i)\},\\
\mathsf{nmaR} &= \{(q_0,q_1)c_0,c_1 \xrightarrow{(\rangle_i,\rangle_i)} (q_2,q_3)1,1 \mid q_0 \overset{\rangle_i}{\longmapsto} q_2,\ q_1 \overset{\rangle_i}{\longmapsto} q_3,\ c_0 \vee c_1\}.
\end{aligned}$$

As with $\Pi^R_{\equiv}(G)$, the language of $\Pi^N_{\equiv}(G)$ describes ambiguous path pairs through $\Gamma_{\equiv}(G)$. The difference is that $\mathcal{L}(\Pi^N_{\equiv}(G))$ does not include path pairs without conflicting reductions. Therefore $\mathcal{L}(\Pi^N_{\equiv}(G)) \subseteq \mathcal{L}(\Pi^R_{\equiv}(G))$. Nevertheless, $\Pi^N_{\equiv}(G)$ does at least describe all the *core* parse trees in $\mathcal{C}^a(G)$:

Theorem 3. $\forall \alpha_b, \beta_b \in \mathcal{C}^a(G) : \alpha_b \neq \beta_b \wedge h(\alpha_b) = h(\beta_b) \Rightarrow (\alpha_b, \beta_b) \in \mathcal{L}(\Pi^N_{\equiv}(G))$.

The Theorem shows that if G is ambiguous — that is $\mathcal{C}^a(G)$ is non-empty — $\mathcal{L}(\Pi^N_{\equiv}(G))$ is also non-empty. This means that if $\mathcal{L}(\Pi^N_{\equiv}(G))$ is empty, G is unambiguous.

6.3 Effects on Filtering Parse Trees and Production Rules

The new nma relation enables our parse tree identification algorithm of Section 4 to potentially identify a larger set of irrelevant parse trees, namely $\mathcal{C}^u(G)$. These trees might be ambiguous, but this is not a problem because we are interested in finding the trees of the smallest possible sentential forms of G, namely the ones in $\mathcal{C}^a(G)$.

Definition 16. *Given $Q^u_{\equiv}$ from $\Pi^N_{\equiv}(G)$, the set of parse trees not in the ambiguous core of G, identifiable with $\equiv$, is $\mathcal{C}^u_{\equiv}(G) = \{\alpha_b\beta_b \mid \exists q \in Q^u_{\equiv}, \alpha_b \bullet \beta_b \in q\}$.*

Theorem 4. *For all equivalence relations $\equiv$, $\mathcal{C}^u_{\equiv}(G) \subseteq \mathcal{C}^u(G)$.*

The set of harmless production rules that can be identified with $\Pi^N_{\equiv}(G)$ is also potentially larger. It might include rules that can be used in parse trees of ambiguous strings, but not in parse trees in $\mathcal{C}^a(G)$. Therefore they are not vital for the ambiguity of G.

Definition 17. *Given $Q^a_{\equiv}$ and $I^a_{\equiv}$ from $\Pi^N_{\equiv}(G)$, the set of harmless productions of G, identifiable from $\Pi^N_{\equiv}(G)$, is $P'_{\mathrm{hl}} = P \setminus \{A \to \alpha \mid \mathsf{proditems}(A \to \alpha) \subseteq I^a_{\equiv}\}$.*

Theorem 5. $\forall p \in P'_{\mathrm{hl}} : \neg\exists \alpha_b \langle_i \beta_b \rangle_i \gamma_b \in \mathcal{C}^a(G) : i = \mathsf{pid}(p)$.

7 Excluding Parse Trees Iteratively

Our approach for the identification of parse trees of unambiguous strings is most useful if applied in an iterative setting. By checking the remainder of the potentially ambiguous parse trees again, there is possibly less interference of the trees during approximation. This could result in less ambiguous path pairs in the position pair automaton. We could then exclude a larger set of parse trees and production rules.

Example 6. The grammar below ([4]) is unambiguous but needs two iterations of the NU Test with item_0 to detect this. At first, $\Pi^N_{\mathsf{item}_0}(G)$ contains only the ambiguous path pair $\langle_1\langle_4 c\rangle_4\rangle_1$ and $\langle_2\langle_5\langle_6 c\rangle_6\rangle_3\rangle_1$. The first path describes a valid parse tree, but the second does not. From $B \to \bullet Cb$ it derives to $C \to \bullet c$, but

Table 1. Excerpt from Results of prototype implementation

Grammar		Harmless rules			Time Amber		Time CfgAnalyzer	
Name	Rules	LR(0)	SLR(1)	LR(1)	Original	Filtered	Original	Filtered
SQL.1	79	65	65	65	28m26s	0.1s	17.6s	1.8s
Pascal.3	176	21	30	144	31.8s	0.0s	9.6s	1.3s
C.2	212	41	44	44	4.5h[1]	4.12s[1]	3.0h	1.1h
Java.1	349	56	70	74	25.0h[2]	22m52s[2]	48.9s	32.4s

[1]for sentences of length 7 (first ambiguity at length 13)
[2]for sentences of length 12 (first ambiguity at length 13)

from $C \to c\bullet$ it reduces to $A \to aC\bullet$. Therefore productions 2, 5 and 3 are only used partially, and they are thus harmless. After removing them and checking the reconstructed grammar again there are no ambiguous path pairs anymore.

$$1 : S \to A,\ \ 2 : S \to B,\ \ 3 : A \to aC,\ \ 4 : A \to c,\ \ 5 : B \to Cb,\ \ 6 : C \to c \qquad (4)$$

We can gain even higher precision by choosing a new equivalence relation with each iteration. If with each step $\Gamma_{\equiv}(G)$ better approximates $\mathcal{S}(G_b)$, we might end up with only the parse trees in $\mathcal{P}^u(G)$. Unfortunately, the ambiguity problem is undecidable, and this process does not necessarily have to terminate. There might be an infinite number of equivalence relations that yield a finite number of equivalence classes. Or at some point we might need to resort to equivalence relations that do not yield a finite graph. Therefore, the iteration has to stop at a certain moment, and we can continue with an exhaustive search of the remaining parse trees.

In the end this exhaustive searching is the most practical, because it can point out the exact parse trees of ambiguous strings. A drawback of this approach is its exponential complexity. Nevertheless, excluding sets of parse trees beforehand can reduce its search space significantly, as we see in the next section.

8 Prototype Results

In [3] we tested a prototype implementation of our approach on a collection of programming language grammars. From unambiguous grammars of SQL, Pascal, C and Java, we created 5 ambiguous versions for each language. For each grammar we tested the number of harmless production rules we could find with the NU Test, using different equivalence relations. Columns 3-5 of Table 1 show the results of these tests for a selection of 4 ambiguous grammars. Similar numbers of harmless rules could be found for the other grammars.

Columns 7-9 show the effect that the removal of the harmless productions had on the run-time of the two exhaustive derivation generators Amber [13] and CfgAnalyzer [1]. They mention the time needed to find the first ambiguous derivation of a grammar before and after filtering with LR(1). We see significant reductions in run-time, sometimes orders of magnitude. For the other grammars we witnessed similar effects.

9 Conclusions

We showed how the Regular Unambiguity Test and Noncanonical Unambiguity Test can be extended to conservatively identify parse trees of unambiguous strings. From these trees we can identify production rules that do not contribute to the ambiguity of the grammar. This information is already very useful for a grammar developer, but it can also be used to significantly reduce the search space of other ambiguity detection methods.

References

1. Axelsson, R., Heljanko, K., Lange, M.: Analyzing context-free grammars using an incremental SAT solver. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 410–422. Springer, Heidelberg (2008)
2. Basten, H.J.S.: Tracking down the origins of ambiguity in context-free grammars. Tech. Rep. SEN-1005, CWI, Amsterdam, The Netherlands (2010)
3. Basten, H.J.S., Vinju, J.J.: Faster ambiguity detection by grammar filtering. In: Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications (LDTA 2010). ACM, New York (2010)
4. Brabrand, C., Giegerich, R., Møller, A.: Analyzing ambiguity of context-free grammars. Science of Computer Programming 75(3), 176–191 (2010)
5. Cantor, D.G.: On the ambiguity problem of Backus systems. Journal of the ACM 9(4), 477–479 (1962)
6. Chomsky, N., Schützenberger, M.: The algebraic theory of context-free languages. In: Braffort, P. (ed.) Computer Programming and Formal Systems, pp. 118–161. North-Holland, Amsterdam (1963)
7. Floyd, R.W.: On ambiguity in phrase structure languages. Communications of the ACM 5(10), 526–534 (1962)
8. Ginsburg, S., Harrison, M.A.: Bracketed context-free languages. Journal of Computer and System Sciences 1(1), 1–23 (1967)
9. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (1979)
10. Knuth, D.E.: On the translation of languages from left to right. Information and Control 8(6), 607–639 (1965)
11. Schmitz, S.: Conservative ambiguity detection in context-free grammars. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 692–703. Springer, Heidelberg (2007)
12. Schmitz, S.: An experimental ambiguity detection tool. Science of Computer Programming 75(1-2), 71–84 (2010)
13. Schröer, F.W.: AMBER, an ambiguity checker for context-free grammars. Tech. rep., compilertools.net (2001), http://accent.compilertools.net/Amber.html
14. Sippu, S., Soisalon-Soininen, E.: Parsing theory. Languages and parsing, vol. 1. Springer, New York (1988)

Prioritized *slotted-Circus*[*]

Paweł Gancarski and Andrew Butterfield

Lero@TCD, Trinity College Dublin
{gawcarsp,butrfeld}@tcd.ie

Abstract. This paper describes an extension adding priority to *slotted-Circus*, a generic framework for reasoning about discretely timed and/or synchronously clocked systems. The semantics of prioritised external choice is given using the Unifying Theories of Programming framework (UTP). The resulting language is similar to Prioritized Timed CSP, but its semantics is not based on trace ordering, and neither does it use the notion of acceptances (e.g. PCSP). Instead, the semantics is based on the notion of refusal sets already widely used in theories of CSP, *Circus* and *slotted-Circus*. We introduce priority as a lightweight extension of *slotted-Circus*, which can be easily adapted to define a similar extension of Timed CSP. We also discuss why priority can most easily be added to specific history models, and the fact that requiring the clock to tick after every communication event results in a more tractable theory.

1 Introduction

1.1 Prioritized slotted-Circus

The formal notation *Circus* is an unification of Z and CSP, to give a "state-rich" process algebra with (restricted) global shared variables. *Circus* has been given a semantics in UTP [OCW09]. Apart from event sequencing, there is no notion of time in *Circus*. A timed version of *Circus* (*Circus* Time Action or CTA) has been explored [SH02, She06] that introduces the notion of discrete time-slots in which sequences of events occur. The semantics of CTA has been developed using UTP, and there we find a two-level notion of history: the top-level views history as a sequence of time-slots; whilst the bottom-level records a history of events within a given slot.

Our interest in hardware compilation languages such as Handel-C [Cel02] led to a development of a generic theory (called *slotted-Circus*), with time-slots whose bottom-level contents could be parameterised, as simple traces, or multisets of events, or as one of the three successively more complex "micro-slot" structures [BSW07]. *slotted-Circus* has also been given a semantics in UTP [GB09].

One important aspect of hardware compilation languages, namely priority, has not been addressed before in Circus based languages. Priority is a very basic

[*] This research was supported by grant 07-RFP-CMSF186 from Science Foundation Ireland, as well as partial support from Lero, the Irish Software Engineering Research Centre.

and intuitively simple concept, used to express that one thing is regarded as more important than another. Even though in the original CSP, priority was not defined, many languages that adopted the CSP model of communication added priority constructs (occam , Ada, Handel-C), usually as they provide for efficient implementations. Even though in this work we are primarily interested in priority for *slotted-Circus* we will also discuss it in the context of CSP. Because this work is focused on defining a theory for hardware description languages, we will only be interested in defining prioritized versions of external choice ($\overleftarrow{\Box}$) and not in prioritized parallel composition ($\overleftarrow{\parallel}$). This is because hardware can utilise true parallelism, and does not require interleaving of notionally parallel processes, with the attendant need to be able to prioritize aspects of that interleaving (a.k.a. scheduling). One of the interesting aspects of Prioritized *slotted-Circus* is that it is very similar to Prioritized Timed CSP (PTCSP) [Low93]. The two notations have different origins and semantic models, but the degree of convergence in key notions and laws, gives us confidence that our priority concept is correct. In PTCSP we can remove non-determinism from a mixture of parallel composition and external choice by prioritising them both, as per the following law:

$$(a \xrightarrow{n} P \overleftarrow{\Box} a \xrightarrow{n} Q) \overleftarrow{\parallel} (a \xrightarrow{n} Q \overleftarrow{\Box} a \xrightarrow{n} P) = a \xrightarrow{n} P$$

In Prioritised *slotted-Circus*(PSC), the lack of a prioritized parallel construct means that we cannot so easily remove all non-determinism during implementation:

$$(a \xrightarrow{n} P \overleftarrow{\Box} a \xrightarrow{n} Q) \parallel (a \xrightarrow{n} Q \overleftarrow{\Box} a \xrightarrow{n} P) = a \xrightarrow{n} P \sqcap a \xrightarrow{n} Q$$

1.2 UTP: General Principles

Theories in UTP are expressed as second-order predicates[1] over a pre-defined collection of free observation variables, referred to as the *alphabet* of the theory. The predicates are generally used to describe a relation between a before-state and an after-state, the latter typically characterised by dashed versions of the observation variables. A predicate whose free variables are all undashed, referring only to the before-state, is called a *condition*. We note that UTP follows the key principle that "programs are predicates" [Hoa85b] and so does not distinguish between the syntax of some language and its semantics as alphabetised predicates. For example, if *ok* denotes absence of divergence, *tr* is a sequence of observed events and *ref* is a set of events currently being refused, then

$$ok' \wedge tr' = tr \frown \{a\} \wedge a \notin ref'$$

denotes an observation of a non-divergent process execution that has performed an a-event and is still willing to perform more. A given theory is characterised by its alphabet, and a series of *healthiness conditions* that constrain the valid assertions that predicates may make. A healthiness condition is a property of a

[1] Most definitions are in fact 1st-order, but we need 2nd-order in order to handle the notion of "healthiness", and recursion.

predicate that distinguishes sensible predicates from nonsense. So, for example the following predicate is clearly nonsense under our intended interpretation:

$$tr = tr' \frown \langle a \rangle$$

It asserts that the history of events that had occurred before this process started (tr) is longer than the history at the end (tr'). It can be ruled out by the following healthiness condition:

$$P \Rightarrow tr \leq tr'$$

Note that healthiness conditions should not be confused with ordinary conditions (predicates with only before-variables).

1.3 Structure and Focus

We first present an overview of *slotted-Circus* semantics §2, before giving an extensive exposition of the prioritised theory in §3. We then wrap up by discussing related §4 and future §5 work, and concluding §6.

2 slotted-Circus

In this section we will focus only on aspects of *slotted-Circus* relevant to this paper. More detailed definitions and explanations can be found in [BG09, GB09].

2.1 Syntax

The syntax of Slotted-*Circus* is similar to that of *Circus*, and a subset, relevant to this paper, is shown in Figure 1. Apart from assignment, we shall ignore the imperative state aspects of the theory as these are covered elsewhere and not relevant to the topic of priority. The basic actions *Skip*, *Stop*, *Chaos*, as well as event prefix ($e \rightarrow A$) and hiding ($A \setminus H$) are similar to the corresponding CSP behaviours [Hoa85a, Sch00], while we also introduce variable assignmement (:=). Actions can be combined with internal ($\sqcap$) or external ($\Box$) choice, sequential composition (;), or parallel composition ($\|$). The key construct related to time-slots, and hence not part of *Circus*, is *Wait t*, which denotes an action that simply waits for t time-slots to elapse, and then terminates.

$$\begin{array}{rcl}
\mathsf{Action} & ::= & \mathit{Skip} \mid \mathit{Stop} \mid \mathit{Chaos} \mid \mathit{Wait}\ \mathsf{t} \mid \mathsf{Name} := \mathsf{Expr} \\
 & \mid & \mathsf{Comm} \rightarrow \mathsf{Action} \mid \mathsf{Action} \sqcap \mathsf{Action} \mid \mathsf{Action} \Box \mathsf{Action} \\
 & \mid & \mathsf{Action}\ ;\ \mathsf{Action} \mid \mathsf{Action} \parallel \mathsf{Action} \mid \mathsf{Action} \setminus \mathsf{CS} \\
\mathsf{Comm} & ::= & \mathsf{Name.Expr} \mid \mathsf{Name!Expr} \mid \mathsf{Name?Name} \\
\mathsf{Expr} & ::= & \text{expression} \\
\mathsf{t} & ::= & \text{positive integer valued expression} \\
\mathsf{Name} & ::= & \text{channel or variable names} \\
\mathsf{CS} & ::= & \text{channel name sets}
\end{array}$$

Fig. 1. Slotted-*Circus* Syntax

2.2 History Models

In *slotted-Circus* a trace model is built on top of exchangeable history models. It is recorded as a sequence of slots, where every slot is defined as a pair consisting of an event history (*hist*) and a refusal set (*ref*), meaning that in the relevant time-slot that event-history *hist* occurred, with events in *ref* being refused afterwards.

$$\mathcal{S}\,E \mathrel{\widehat=} \mathcal{H}\,E \times \mathbb{P}\,E$$

There are currently two history models defined and working within the *slotted-Circus* framework: MSA and CTA. In CTA, a history is just a sequence ("trace") of events in the order in which they occurred during a slot. So the following example shows a run of CTA:

$$\langle(\langle\rangle, ref_1), (\langle a, b\rangle, ref_2), (\langle b, a, a\rangle, ref_3), (\langle b, a\rangle, ref_4), \ldots\rangle$$

In the multi-set action (MSA) variant, we ignore event ordering within slots, viewing histories as a bag of events, so the above example appears as:

$$\langle(\{\}, ref_1), (\{a \mapsto 1, b \mapsto 1\}, ref_2), (\{a \mapsto 2, b \mapsto 1\}, ref_3), (\{a \mapsto 1, b \mapsto 1\}, ref_4), \ldots\rangle$$

The MSA history model carries no information on event ordering within time-slots.

2.3 UTP Observations

In our UTP theory, we model *slotted-Circus* using four observations:

$ok : \mathbb{B}$ — stability, absence of serious error.
$wait : \mathbb{B}$ — waiting, true if process is waiting on events, false if it has terminated.
$slots : (\mathcal{S}\,E)^+$ — full event history as a non-empty sequence of slots, with "clock-ticks" occurring at the boundaries between slots.
$state : Variable \rightarrow\!\!\!\!\!+ \; Value$ — an environment giving program variable values.

The alphabet of our theory consists of the above four variables representing the state before an action starts, and dashed versions of the variables giving the (current/final) state when an action is running and either waiting for an event or just terminated.

2.4 Healthiness Conditions

Healthiness conditions are characterised by idempotent predicate transformers, with a healthy predicate being a fixed point of such a transformer. Here we shall only consider **R3**, **CSP1,2,3,4** as they are explicitly invoked. **R1** and **R2** deal with the infeasibility of time travel and (direct) memory of past events, and are well covered elsewhere, and satisfied in any case by all definitions we present.

The healthiness condition **R3** is one associated with all "reactive" systems in the UTP, covering process-algebras like ACP, CSP, and CCS.

$$\mathbf{R3}(P) \mathrel{\widehat=} I\!I \lhd wait \rhd P$$

R3 deals with the situation when a process has not actually started to run, because a prior process has yet to terminate, characterised by $wait = \textsc{True}$. In this case the action of a yet-to-be started process should simply be to do nothing, an action we call "reactive-skip" (II).

A process is **CSP1** healthy if *all* it asserts, when started in an unstable state (due to some serious earlier failure), is that the event history may be extended:

$$\mathbf{CSP1}(P) \mathrel{\widehat=} P \lor \neg\, ok \land slots \preccurlyeq slots'$$

A process predicate is **CSP2** healthy if it does not mandate instability, so if true with $ok' = False$, it is also true with $ok' = True$, all other observation variables being unchanged.

$$\mathbf{CSP2}(P) \mathrel{\widehat=} P;\ (ok \Rightarrow ok') \land wait' = wait \land slots' = slots \land state' = state$$

CSP3 and **CSP4** state respectively that *Skip* is a left and right unit of sequential composition.

$$\mathbf{CSP3}(P) \mathrel{\widehat=} Skip;\ P \qquad\qquad \mathbf{CSP4}(P) \mathrel{\widehat=} P;\ Skip$$

A more technical aspect of **CSP3,4** is that once *Skip* starts/terminates it unconstrains the refusals set of the last slot. This causes **CSP3** to make processes insensitive to refusals of a previously terminated process, and **CSP4** to unconstrain refusals of a processes once they terminate. These properties are changed in our prioritized model, allowing information about refusals to be partially propagated even when a process terminates.

2.5 Slotted Semantics

The language constructs of sequential composition and internal choice all have the same semantics as in standard UTP:

$$\begin{aligned} P;\ Q &\mathrel{\widehat=} \exists\, obs_m \bullet P[obs_m/obs'] \land Q[obs_m/obs] \\ P \sqcap Q &\mathrel{\widehat=} P \lor Q \end{aligned}$$

Here *obs* is shorthand for all the observational variables.

Semantic Building Blocks. We define the semantics of *slotted-Circus* in terms of a number of basic predicate building-blocks, largely to do with events and communication, that we now describe informally. The building blocks are all **R1**-, **R2**-healthy, but in general will not satisfy **R3** or the CSP healthiness conditions in themselves— they are intended to be used in constructions that do.

NOEVTS describes a situation that allows time to pass ($\#slots' > \#slots$) but disallows the occurrence of any events (all slot histories are empty).

FSTEVTS(E) asserts that a given set of events (E) have occurred immediately (in the first time slot).

IMMEVTS describes a situation when some events occur immediately (in the first slot). It can be defined as $\exists\, E \bullet E \neq \emptyset \land FSTEVTS(E)$.

$$\begin{aligned}
\mathit{Chaos} &\mathrel{\widehat=} \mathbf{R}(\mathbf{true})\\
\mathit{Miracle} &\mathrel{\widehat=} \mathbf{CSP1}(\mathbf{R3}(\mathit{FALSE}))\\
\mathit{Stop} &\mathrel{\widehat=} \mathbf{CSP1}(\mathbf{R3}(ok' \wedge wait' \wedge \mathit{NOEVTS}))\\
\mathit{Skip} &\mathrel{\widehat=} \mathbf{R3}(\mathbf{CSP1}(state = state' \wedge \neg wait' \wedge ok' \wedge slots \cong slots'))\\
\mathit{Wait}\ t &\mathrel{\widehat=} \mathit{Stop}, \qquad \text{if } len(slots' - slots) < t\\
&\phantom{\mathrel{\widehat=}} \mathit{Skip}, \qquad \text{if } len(slots' - slots) = t\\
c!e \rightarrow \mathit{Skip} &\mathrel{\widehat=} c.e \rightarrow \mathit{Skip}\\
c \rightarrow A &\mathrel{\widehat=} (c \rightarrow \mathit{Skip});\ A\\
c?x \rightarrow \mathit{Skip} &\mathrel{\widehat=} \mathop{\Box}_{k:T} \bullet (c.k \rightarrow \mathit{Skip};\ x := k)
\end{aligned}$$

Fig. 2. Semantics of basic actions

Semantics of Basic Actions. The semantics of the basic actions are described in Fig.2. The worst possible action in *slotted-Circus* is *Chaos*. It is the most unpredictable healthy process, and bottom of the refinement lattice. Action *Miracle* is the top of the refinement lattice, and is a program satisfying any specification (clearly infeasible), but useful as a unit of nondeterministic choice ($\mathit{Miracle} \sqcap P = P$). Action *Stop* has deadlocked: it is stable, never terminates and never performs any event. Action *Skip* terminates immediately in a stable state, without performing any events. In keeping with the CSP definition, *Skip* ignores the refusals of any preceding process, hence the use of slot-equivalence ($\cong$) here, which is slightly weaker than slot-equality, in that it ignores the refusals in the last slot. The action that introduces explicit timed behavior is *Wait* t. It never performs any events and has only two possible behaviors. The first one is to wait for t clock ticks, the second to terminate when the right time is reached.

Event prefix ($c \rightarrow A$) is defined using $c \rightarrow \mathit{Skip}$ composed with A. Output prefixes are simply event prefixes, whilst input prefixes are modelled as an external choice over all possible input values, with assignment being used to capture the outcome. The process $c \rightarrow \mathit{Skip}$ is defined using two basic actions:

$\mathit{WTC}(c)$ allows time to pass without any events occurring, while never refusing to perform event c.

$\mathit{TRMC}(c)$ performs event c in the first time-slot.

Basically while non-terminated, $c \rightarrow \mathit{Skip}$ acts like WTC, and once event c occurs (if at all), it then has the behaviour $\mathit{WTC}(c);\ \mathit{TRMC}(c)$ — waiting followed by event c and termination:

$$c \rightarrow \mathit{Skip} \mathrel{\widehat=} \mathbf{CSP1}\left(ok' \wedge \mathbf{R3}\left(\mathit{WTC}(c) \lhd wait' \rhd \begin{pmatrix} state' = state \wedge \\ \mathit{WTC}(c);\ \mathit{TRMC}(c)\end{pmatrix}\right)\right)$$

Semantics of Composite Actions. External choice ($A \Box B$) allows external events to determine which action runs, so for example if we have $(a \rightarrow A) \Box$

$(b \rightarrow B)$, then, if the environment performs a, we see that event occur, followed by an execution of action A. Unfortunately, the very simple definition[2] of external choice proposed in [HH98] no longer suffices, as we may have to wait for several clock-ticks before an external event arises that resolves the choice.

$$A \,\square\, B \mathrel{\widehat{=}} \mathbf{CSP2}(\,\mathit{Stop} \wedge A \wedge B \;\vee\; \mathit{Choice}(A,B) \;\vee\; \mathit{Choice}(B,A)\,)$$
$$\mathit{Choice}(C,R) \mathrel{\widehat{=}} C \wedge \left(R \wedge \mathit{NOEVTS};\ \begin{pmatrix} \mathit{IMMEVTS} \vee \\ \mathit{slots} \cong \mathit{slots}' \wedge (\neg \mathit{wait}' \vee \neg \mathit{ok}') \end{pmatrix}\right)$$

Predicate $\mathit{Choice}(C,R)$ describes the circumstances where action C has been chosen, whilst R has been refused, which occurs in situations where R has performed no events. We capture these cases as follows: conjoin R with NOEVTS, and follow it sequentially with some "end"-condition E. All of this is conjoined with C to give

$$C \wedge (R \wedge \mathit{NOEVTS};\ E)$$

i.e an execution of C consistent with R having done no events, and then ending in the situation described by E.

Now we can characterise three possible cases were C either: (i) performs an event after a delay: $E = \mathit{IMMEVTS}$; (ii) terminates without performing any events: $E = \mathit{slots} \cong \mathit{slots}' \wedge \neg \mathit{wait}'$ or (iii) diverges but performs no event: $E = \mathit{slots} \cong \mathit{slots}' \wedge \neg \mathit{ok}'$.

The parallel composition $A \parallel B$ runs A and B in lock-step parallel (clock ticks at same time for both). Both actions run on local copies of the variables. The construct terminates when both actions have terminated — if one ends early then its behaviour is padded out with empty slots.

$$A \setminus H \mathrel{\widehat{=}} \mathbf{R3}\begin{pmatrix} \exists\, s' \bullet A[s'/\mathit{slots}'] \wedge \\ \qquad \mathit{slots}' \fallingdotseq \mathit{slots} = \mathit{map}(\mathit{SHide}(H))(s' \fallingdotseq \mathit{slots}) \\ \quad \wedge\, H \subseteq \bigcap \mathit{Refs}(s' \fallingdotseq \mathit{slots}) \end{pmatrix};\ \mathit{Skip}$$

The hiding operator $A \setminus H$ denotes an execution of action A, but with any events in event-set H hidden. Function $\mathit{SHide}(H)$ removes events in H from a slots history component, and adds them into the refusal set. We enforce a key property of hiding, namely that of *maximal progress*, i.e. hidden events occur as soon as they are enabled. Without this semantic feature the following undesirable law would hold:

$$(a \rightarrow \mathit{Skip}) \setminus \{a\} = \mathit{Wait}\,0 \sqcap \mathit{Wait}\,1 \sqcap \ldots \sqcap \mathit{Wait}\,n \sqcap \ldots$$

This law is undesirable because it makes the performance of a single hidden event followed by termination equal to a wait for an arbitrary number of clock cycles — effectively a weak form of livelock. By forcing hidden events to be refused during every slot, we prevent them from waiting for a clock-tick, because the definition of prefix action requires events not to be refused when waiting. This results in the desired law, namely $(a \rightarrow \mathit{Skip}) \setminus \{a\} = \mathit{Skip}$. At the end we add *Skip* to unconstrain the refusals of the last slot.

[2] $A \,\square\, B \mathrel{\widehat{=}} A \wedge B \lhd \mathit{Stop} \rhd A \vee B$.

3 Prioritized slotted-Circus

As mentioned in the introduction, prioritized choice has been added to CSP inspired programming languages with concurrency, like occam and Ada. These languages were designed on foot of formal semantics covering most language constructs [Ros84], but with priority being an exception. The approach then taken instead was to have a semantics for external choice, or its equivalent, and then consider the prioritised forms to be refinements of the non-prioritised versions, thus

$$(P \Box Q) \sqsubseteq (P \overleftarrow{\Box} Q)$$

justified their use in implementations. We note semantics for these languages that covered priority did emerge afterwards — e.g. [Cam89]. In hardware description languages, priority is very important as implementations need to be deterministic and priority is an effective and efficient way of achieving this. Indeed, in Handel-C, the only form of external choice is prioritised, appearing as the so-called "prialt" language construct. Our aim is to extend the work *slotted-Circus* to cover this important feature.

3.1 Prioritized CSP

The difficulty in formalizing priority is the need for a clear idea of when it gets resolved, i.e. how long should we wait for possible options before we try to choose the one we prefer most? The problems appear in CSP because it is impossible to say how much time has passed, but it is possible to determine event ordering. The problem can be best described using two laws:

$$\begin{aligned} Stop \Box P &= \quad P \\ Skip \Box P &= Skip \sqcap \begin{cases} P, & \text{if P terminates or performs an event} \\ Miracle, & \text{otherwise} \end{cases} \end{aligned}$$

Now, if we consider be the consequence of the laws above after prioritizing the choice we get:

$$Stop \overleftarrow{\Box} P = P$$

Prioritized choice is an implementation of external choice, so no other outcome is possible. However, with *Skip*, because the termination event is immediately available and beats any other external event, we expect to see:

$$Skip \overleftarrow{\Box} P = Skip$$

Because in CSP we can not measure the passing time, the following law holds:

$$Q \setminus Events = \begin{cases} Skip, & \text{if Q terminates} \\ Stop, & \text{otherwise} \end{cases}$$

for that reason, when we combine our results, we get:

$$(Q \setminus Events) \overleftarrow{\Box} P = \begin{cases} Skip, & \text{if Q terminates} \\ P, & \text{otherwise} \end{cases}$$

So, any implementation of prioritized choice would have to be able to solve the Halting problem.

3.2 Zero-Causality Problems

The problem in defining priority for CSP is that we have no finite deadline for the choice to be resolved. In *slotted-Circus*, by contrast, this problem seems to be solved —the deadline is a clock tick. In other words we expect prioritized choice to deal with a situation when two processes are ready to perform an event within the same time slot. Unfortunately a new problem has appeared. However we also have a property from the semantics of *slotted-Circus* of "zero-causality", denoting the fact that communication/events take no time. Because of the maximal urgency principle and zero-causality, if we use hiding on events performed one after another ($(a \rightarrow b \rightarrow \mathit{Skip}) \setminus \{a, b\}$), from one perspective we might say that they are performed at the same time (because within the same time slot), but on the other hand we know that an event b occurred after event a. This fact arises separate problems for both CTA and MSA.

In CTA the problem becomes visible once the prioritized choice operator has been defined.

$$((a \rightarrow b \rightarrow H \overleftarrow{\Box} b \rightarrow a \rightarrow L) \parallel (b \rightarrow a \rightarrow S)) \setminus \{a, b\} =_{CTA} \mathit{Miracle}$$

In CTA we get a miracle, because of a conflict between CTA and priority. There is an tension between the history model in CTA that asserts that order matters, and the semantics we require for priority that waits to the end of the slot to sort everything out. This results intenrally in a contradiction in CTA, leading to the miracle. Creating a definition of priority that would avoid the contradiction emerging here proved to be very difficult, requiring large changes in the semantics and for that reason was abandoned.

The MSA history model behaves properly with prioritized choice:

$$((a \rightarrow b \rightarrow H \overleftarrow{\Box} b \rightarrow a \rightarrow L) \parallel (b \rightarrow a \rightarrow S)) \setminus \{a, b\} =_{MSA} (H \parallel S) \setminus \{a, b\}$$

However, the example above and the idea of zero-causality, led us to the discovery of strange properties unnoticed before in *slotted-Circus*. The first one is a violation of prefix closure:

$$(a \rightarrow b \rightarrow \mathit{Skip}) \parallel (b \rightarrow a \rightarrow \mathit{Skip})$$

The described action can perform a and b at the same time, but can never perform either a or b alone. Another issue with the example above is that the maximal urgency property of hiding no longer holds:

$$((a \rightarrow b \rightarrow \mathit{Skip}) \parallel (b \rightarrow a \rightarrow \mathit{Skip})) \setminus \{a, b\} =_{MSA} \bigsqcap_{n \in N} \mathit{Wait}\ n \sqcap \mathit{Stop}$$

Things get more complicated if we try to "crossover" some information.

$$\begin{pmatrix} (a?x \rightarrow b!x \rightarrow \mathit{Skip}) \\ \parallel (b?x \rightarrow a!x \rightarrow \mathit{Skip}) \end{pmatrix} \setminus \{a, b\} =_{MSA} \bigsqcap_{n \in N} (\mathit{Wait}\ n;\ x :=?) \sqcap \mathit{Stop}$$

$$\begin{pmatrix} (a?x \rightarrow b!x \rightarrow \mathit{Skip}) \\ \parallel (b?x \rightarrow a!(x+1) \rightarrow \mathit{Skip}) \end{pmatrix} \setminus \{a, b\} =_{MSA} \mathit{Stop}$$

3.3 Timed Prefix

In order to deal with the problems presented above, it was decided to remove zero-causality from the language, by stipulating that communication always takes time. This was by the introduction of a new action — timed prefix:

$$a \xrightarrow{n} P =_{def} a \rightarrow \mathit{Wait}\ n;\ P$$

A similar assumption has been made in Prioritized Timed CSP and Handel-C.

So far no other problems of interaction between CTA and the prioritized model have been found, but because the notion of time assumed by priority better suits MSA our research is now focused on supporting this history model. It is still an open question what is the difference (if any), between *slotted-Circus* with CTA or MSA history model, when only timed communication is permitted. Another open problem is the existence of healthiness conditions, implying that communication takes time. So far our research leads us to suggest that it will be a special case of prefix closure, but no prefix closure healthiness condition has yet been defined in UTP theories.

3.4 Defining Priority

In the timed theory, prioritized external choice is very similar in behaviour to external choice. In both cases the choice is resolved on a first-come first-served basis.

$$(a \xrightarrow{n} A \Box (\mathit{Wait}1;\ b \xrightarrow{n} B)) \setminus \{a, b\} = \mathit{Wait}\ n;\ A \setminus \{a, b\}$$
$$(a \xrightarrow{n} A \overleftarrow{\Box} (\mathit{Wait}1;\ b \xrightarrow{n} B)) \setminus \{a, b\} = \mathit{Wait}\ n;\ A \setminus \{a, b\}$$

The differences become visible when an event (or termination) is available for both options at the same time. In that case the external choice becomes nondeterministic:

$$(a \xrightarrow{n} A \Box (b \xrightarrow{n} B)) \setminus \{a, b\} = \mathit{Wait}\ n;\ (A \sqcap B) \setminus \{a, b\}$$

while the prioritized choice chooses the higher priority option:

$$(a \xrightarrow{n} A \overleftarrow{\Box} (b \xrightarrow{n} B)) \setminus \{a, b\} = \mathit{Wait}\ n;\ A \setminus \{a, b\}$$

There are two important facts to observe here. The first one is that prioritized external choice is an implementation (refinement) of the normal one. Which means that any behaviour accepted by it is also accepted by the external choice. The second fact is that the behaviour of the high priority choice is accepted iff it is accepted by the external choice. In other words we can describe priority using an external choice definition and a condition strengthening the low priority option.

$$A \overleftarrow{\Box} B \mathrel{\widehat{=}} (A \wedge B \wedge \mathit{Stop}) \vee \mathit{Choice}(A, B) \vee \mathit{WeakChoice}(B, A)$$

Where

$$\mathit{WeakChoice}(B, A) = \mathit{Choice}(B, A) \wedge \mathit{MagicCondition}$$

The only question left to be answered is: what is “MagicCondition”? To do that we need to bear in mind the following important laws:

$$\begin{array}{ll}
(1) & (a \xrightarrow{n} A \overleftarrow{\Box} b \xrightarrow{n} B) \setminus \{a, b\} = \mathit{Wait}\ n;\ A \setminus \{a, b\} \\
(2) & a \xrightarrow{n} A \overleftarrow{\Box} a \xrightarrow{n} B = a \xrightarrow{n} A \\
(3) & \mathit{Skip} \overleftarrow{\Box} A = \mathit{Skip} \\
(4) & a \xrightarrow{n} A \overleftarrow{\Box} b \xrightarrow{n} B \neq a \xrightarrow{n} A, \qquad a \neq b \\
(5) & (P \Box Q) \sqsubseteq (P \overleftarrow{\Box} Q)
\end{array}$$

The first three laws describe the difference in behaviour in comparison with external choice. Law (1) shows us a typical use case for priority, where two racing processes want to perform an event at exactly the same time. In that case hiding makes both of the events internal and they both are only willing to perform an event at the first time slot. In law (2) we can see a situation when we do not know when an event is going to be performed, but only that both of the racing processes will perform an event at the same time (because it is exactly the same event). For that reason the high priority option will always be chosen. Law (3) states that a termination is treated as an event and can resolve a prioritized choice. Finally law (4) is in contrast to both law (1) and (2) and it makes sure that the ”MagicCondition” is not too strong.

WeakChoice can be best describe by contrast to the *Choice* definition.

$$\mathit{Choice}(P, S) \mathrel{\widehat{=}} \mathit{CSP2}(P \wedge \left(\begin{array}{l} (S \wedge \mathit{NOEVTS}); \\ \left(\begin{array}{l} \mathit{IMMEVTS} \\ \vee\ \mathit{slots} \cong \mathit{slots}' \wedge (\neg \mathit{wait}' \vee \neg \mathit{ok}') \end{array} \right) \end{array} \right))$$

The simplest law to address is (3). To make it hold we only need to add a *wait'* clause in the right place of the *Choice* definition.

$$\mathit{CSP2}(\mathit{LP} \wedge \left(\begin{array}{l} (\mathit{HP} \wedge \mathit{NOEVTS} \wedge \mathbf{wait}'); \\ \left(\begin{array}{l} \mathit{IMMEVTS} \\ \vee\ \mathit{slots} \cong \mathit{slots}' \wedge (\neg \mathit{wait}' \vee \neg \mathit{ok}') \end{array} \right) \end{array} \right))$$

That way we make sure that the LP (low priority option) behaviour can only be chosen when the HP (high priority option) at the time of resolution is in a waiting state. The second law is addressed by changing *IMMEVTS* into:

$$\exists\, E \bullet \mathit{FSTEVTS}(E) \wedge E \neq \emptyset \wedge E \subseteq \mathit{sref}(\mathit{last}(\mathit{slots}))$$

That way LP can only perform an event which is refused by HP. Finally we can address the remaining laws - (1) and (4). According to those laws, LP can only be refused when HP is willing to perform an event and the environment is demanding the event to be performed in the first possible time slot. This behaviour is very similar to the waiting option of the prefix operator:

$$\mathbf{CSP1}(ok' \wedge \mathbf{R3}(WTC(c) \wedge wait')) \setminus \{c\} = Miracle$$

that gets refused when the environment decides that an event needs to be performed in the first time slot. For that reason we want to copy the mechanism of interaction between hiding and prefix, by not refusing events that are not refused by HP at the time when the choice is being resolved:

$$\mathbf{CSP2}(LP \wedge \left(\begin{array}{l} (HP \wedge NOEVTS \wedge wait'); \\ \left(\begin{array}{l} \mathbf{sref}(\mathbf{head}(\mathbf{slots'} \setminus\!\!\setminus \mathbf{slots})) \subseteq \mathbf{sref}(\mathbf{last}(\mathbf{slots})) \wedge \\ \left(\begin{array}{l} (\exists\, E \bullet FSTEVTS(E) \wedge E \neq \emptyset \wedge E \subseteq sref(last(slots))) \\ \vee\ slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) \end{array} \right))$$

We then obtain the following law:

$$WeakChoice(LP, a \xrightarrow{n} HP) \setminus \{a\} = Miracle$$

that allows us to prove (1).

3.5 Changes in slotted-Circus

One of the goals, when introducing priority to *slotted-Circus*, was to minimize changes made by the expansion to the semantics of the language. Happily there are only two aspects that need to be addressed: a new healthiness condition and a fix to the semantic definition of hiding.

Because our prioritized choice operator uses refusals sets in a new way and expands their functionality, we have to make sure that information stored in it is properly propagated and whenever a process terminates **CSP4** no longer completely unconstrains its refusals sets. We can do that by introducing a new healthiness condition:

$$\mathbf{PRI}(P) \mathrel{\widehat=} P \wedge (ok \Rightarrow sref(head(slots' \setminus\!\!\setminus slots)) \subseteq sref(tail(slots)))$$

This ensures that whenever a process is not refusing an event then this information will not be omitted by the following processes. Once we make sure that *Skip* is **PRI** healthy, we ensure that **CSP4** can only reduce refusals.

The problem with the semantics of hiding is that it doesn't satisfy one of the healthiness conditions —**CSP3**. While it was not a problem before, e have to be very careful with refusals when we add priority. For that reason we make sure that refusals of a previous process have no influence at the behaviour of hiding by unconstraining them inside the definition ($\exists\, s \bullet slots \cong s$).

$$A \setminus hidn \mathrel{\widehat=} \mathbf{PRI} \circ \mathbf{R3} \left(\begin{array}{l} \exists\, \mathbf{s}, s' \bullet A[\mathbf{s}, s'/slots, slots'] \wedge \\ \qquad \mathbf{slots} \cong \mathbf{s} \wedge hidn \subseteq \bigcap srefs(s' \setminus\!\!\setminus \mathbf{s}) \\ \qquad slots' \setminus\!\!\setminus slots = map(shide(hidn))(s' \setminus\!\!\setminus s) \wedge\ ) \end{array} \right)$$

3.6 Prioritized Choice Definition

After introducing the new healthiness condition we can finally present a complete definition of prioritized choice:

$$\begin{aligned}
L \overrightarrow{\Box} H &\mathrel{\widehat=} H \overleftarrow{\Box} L \\
H \overleftarrow{\Box} L &\mathrel{\widehat=} H \wedge L \wedge \mathit{Stop} \vee \mathit{Choice}(H, L) \vee \mathit{WeakChoice}(L, H) \\
\mathit{WeakChoice}(L, H) &\mathrel{\widehat=} \mathbf{CSP2}(L \wedge \\
&\left(\begin{array}{l} \left(H \wedge \mathit{NOEVTS} \wedge \mathit{wait}' \right); \\ \mathbf{PRI} \left(\begin{array}{l} \exists E \bullet \left(\begin{array}{l} \mathit{FSTEVTS}(E) \\ \wedge E \neq \emptyset \wedge E \subseteq \mathit{sref}(\mathit{tail}(\mathit{slots})) \end{array} \right) \\ \vee \mathit{slots} \cong \mathit{slots}' \wedge (\neg \mathit{wait}' \vee \neg \mathit{ok}') \end{array} \right) \end{array} \right))
\end{aligned}$$

4 Related Work

Work on priority in process algebras can be grouped basically into two main camps: those based on CSP and associated CSP-like languages [Bar89, Fid93, Low93]; and those focussing on labelled transitions systems (the "CCS school) [CH90, CW95, HL98, BG00, CLN07]

In the latter camp, priority has been investigated by adding it as a means for selecting out from many labelled arcs leaving a state, with differences based on how priorities are assigned (local vs. global) [CH90, CW95]. An emphasis has been on characterising relevant bisimulations, congruences and corresponding axiomatizations of priority in these settings [HL98, BG00, CLN07]. Despite the extensive work done on priority in a CCS setting however, it is not the case that priority in process algebras has "been done". When presenting our operational semantics of Handel-C [BW05] we pointed out that we had two notions of priority: one that of the `prialt` construct in the language, the other associated with the LTS we constructed, neither of which corresponded to any of the priority models described in the CCS literature.

Of interest is the BIP system developed by Sifakis and colleagues [BBS06] that views systems as components built in three layers: an LTS with ports to communicate with the outside world; a notion of interaction as a set of ports from different LTSs; and using a priority scheme to select among enabled interactions. This system at an abstract level, is quite similar to both the notion of prioritised choice in Handel-C and the slots concepts that we have formalised in *slotted-Circus*, and it may prove fruitful to investigate the relationship more closely.

In the CSP camp, more emphasis has been placed on the use of denotational semantics. In terms of priority, early work on the use of priority in implementations said very little about its semantics except that prioritised choice was a refinement of external choice. Interesting early exceptions were work giving occam an operation semantics [Bar89, Cam89]. More substantial work on priority in a denotational setting was presented by Colin Fidge [Fid93] introducing the notion of *preferences*. At the same time, Gavin Lowe characterised both probabilistic and prioritised CSP as refinements of Timed CSP, and established linkages between probability and priority [Low93].

5 Future Work

Also worthy of exploration are the details of the behaviour of the Galois links [HH98, Chp 4] between *slotted-Circus*, with and without priority, and between those and standard *Circus*. These details will provide a framework for a comprehensive refinement calculus linking all these reactive theories together. The goal is a scheme whereby *Circus* is a specification language and *slotted-Circus* is a refinement stage, on the way to a hardware implementation, captured in prioritised *slotted-Circus*. We plan to perform some case studies, looking at hardware interfaces for flash memory, as well as exploring wireless network protocols for which our prioritised model seems surprisingly well-suited (we can use priority to capture collision detection).

6 Conclusions

A denotational semantics for prioritised *slotted-Circus* has been presented, and we have shown that prioritized choice in *slotted-Circus* and its laws fit the `prialt` construct in Handel-C. Of particular interest has been the introduction of the "clock-tick after communication" constraint in order to get a sensible theory. We have also identified close linkages between our work and that on prioritised CSP, and extensions to timed CSP.

It is still an open question what is the difference (if any), between the CTA or MSA history models, when only timed communication is permitted. Another open problem is the existence of healthiness conditions, implying that communication takes time. So far our research leads us to suggest that it will be a special case of prefix closure, but no prefix closure healthiness condition has yet been formalised in UTP.

Acknowledgements. We would like to thank Jim Woodcock and his colleagues for many fruitful discussions on various aspects of this work. This work was also partially supported by Lero, the Irish Software Engineering Research Centre, SFI grant 03/CE2/I303_1.

References

[Bar89] Barrett, G.: The semantics of priority and fairness in occam. In: Main, M.G., Melton, A., Mislove, M.W., Schmidt, D.A. (eds.) MFPS 1989. LNCS, vol. 442, pp. 194–208. Springer, Heidelberg (1990)

[BBS06] Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), Pune, India, September 11-15, pp. 3–12. IEEE Computer Society, Los Alamitos (2006)

[BG00] Bravetti, M., Gorrieri, R.: A complete axiomatization for observational congruence of prioritized finite-state behaviors. In: Montanari, U., Rolim, J.D.P., Welzl, E. (eds.) ICALP 2000. LNCS, vol. 1853, pp. 744–755. Springer, Heidelberg (2000)

[BG09] Butterfield, A., Gancarski, P.: Slotted-Circus: A generic UTP framework for discretely-timed *circus*. Technical Report TCD-CS-09-32, School of Computer Science & Statistic Trinity College Dublin, Trinity College, Dublin 2, Ireland (July 2009), `https://www.cs.tcd.ie/publications/tech-reports/reports.09/TCD-CS-2009-32.pdf`

[BSW07] Butterfield, A., Sherif, A., Woodcock, J.: Slotted-*circus*: A UTP-family of reactive theories. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 75–97. Springer, Heidelberg (2007)

[BW05] Butterfield, A., Woodcock, J.: `prialt` in Handel-C: an operational semantics. International Journal on Software Tools for Technology Transfer (STTT) 7(3), 248–267 (2005)

[Cam89] Camilleri, J.: An operational semantics for OCCAM. International Journal of Parallel Programming 18(5), 149–167 (1989)

[Cel02] Celoxica Ltd. Handel-C Language Reference Manual, v3.0 (2002), `http://www.celoxica.com`

[CH90] Cleaveland, R., Hennessy, M.: Priorities in process algebras. Inf. Comput. 87(1/2), 58–77 (1990)

[CLN07] Cleaveland, R., Lüttgen, G., Natarajan, V.: Priority and abstraction in process algebra. Inf. Comput. 205(9), 1426–1458 (2007)

[CW95] Camilleri, J., Winskel, G.: CCS with priority choice. Inf. Comput. 116(1), 26–37 (1995)

[Fid93] Fidge, C.J.: A formal definition of priority in CSP. ACM Transactions on Programming Languages and Systems 15(4), 681–705 (1993)

[GB09] Gancarski, P., Butterfield, A.: The denotational semantics of slotted-circus. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 451–466. Springer, Heidelberg (2009)

[HH98] Hoare, C.A.R., He, J.: Unifying Theories of Programming. Series in Computer Science. Prentice Hall, Englewood Cliffs (1998), `http://www.unifyingtheories.org`

[HL98] Hermanns, H., Lohrey, M.: Priority and maximal progress are completely axiomatisable. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 237–252. Springer, Heidelberg (1998)

[Hoa85a] Hoare, C.A.R.: Communicating Sequential Processes. Intl. Series in Computer Science. Prentice Hall, Englewood Cliffs (1985)

[Hoa85b] Hoare, C.A.R.: Programs are predicates. In: Proc. of a discussion meeting of the Royal Society of London on Mathematical logic and programming languages, pp. 141–155. Prentice-Hall, Inc., Englewood Cliffs (1985)

[Low93] Lowe, G.: Probabilities and Priorities in Timed CSP. PhD thesis, Oxford University (1993)

[OCW09] Oliveira, M., Cavalcanti, A., Woodcock, J.: A UTP semantics for circus. Formal Asp. Comput. 21(1-2), 3–32 (2009)

[Ros84] Roscoe, A.W.: Denotational semantics for occam. In: Brookes, S.D., Roscoe, A.W., Winskel, G. (eds.) Seminar on Concurrency. LNCS, vol. 197, pp. 306–329. Springer, Heidelberg (1985)

[Sch00] Schneider, S.: Concurrent and Real-time Systems — The CSP Approach. Wiley, Chichester (2000)

[SH02] 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)

[She06] Sherif, A.: A Framework for Specification and Validation of Real Time Systems using Circus Action. Ph.d. thesis, Universidade Federale de Pernambuco, Recife, Brazil (January 2006)

A Denotational Semantical Model for Orc Language

Qin Li, Huibiao Zhu, and Jifeng He

Software Engineering Institute, East China Normal University, Shanghai, China
{qli,hbzhu,jifeng}@sei.ecnu.edu.cn

Abstract. Orc language is a concurrency calculus proposed to study the orchestration patterns in wide area computing. Its special properties such as high concurrency and asynchronism makes it a brilliant subject to study the distributed service oriented systems. This paper proposes a denotational semantical model for Orc language. Every Orc program is formalized to a predicate. Healthiness conditions are provided to make the program domain corresponding to a specific subset of predicate domain. This model gives the same semantical interpretation to the implementations and specifications. With the refinement principle, we are able to determine whether a program satisfies its specification, which can be illustrated by theorem provers.

1 Introduction

With the development of the web technology, more and more resources can be accessed through Internet. Web applications work not only relying on the local resources but also require the usage of resources published on Internet. It is a trend to complete a computation through the cooperations of many independent entities. In order to achieve better reusability, software is encapsulated as web services providing independent computation resources to their users [1]. Distributed computation and concurrent orchestration are studied and widely used in modern computation models. In these models, the user's computation task is dispatched to distributed entities according to their capabilities. Users can only invoke the remote computation services without knowing their implementation details. Furthermore, the distribution of services makes the communications among them are always asynchronous. A popular architecture aiming to address the problem is known as Service Oriented Architecture (SOA) [5]. The idea of invoking a published service instead of developing an isolated function leads a revolution of web application development. A business job can be arranged by orchestrating the services which can provide computation resources or functional supports. Along with this trends, researchers begin to focus on the issues on the service orchestration.

The Orc Language, abbreviation of Orchestration Language, is a concurrency calculus firstly designed by J. Misra et al. [9]. The language is proposed to study the orchestration patterns of service-oriented computing [4]. The original Orc calculus has only three combinators. However, they can construct most of orchestration patterns which are usually adopted in practice. In addition, Orc language has the capability to orchestrate services according to their execution status. The Orc programming language has been implemented by A. Quark et al. from the University of Texas, Austin [10]. The programming language has a functional language named Cor as its basis. There is a demo trail of Orc programming language available on the web site [12].

Many researchers have studied the semantics of Orc concurrency calculus. A standard operational semantics of Orc is proposed in [11] by J. Misra et al.. In the further research, I. Wehrman et al. have proved the key algebra laws using the bisimulation equivalence [14]. His work has also given a trace representation for Orc. But D. Vardoulakis et al. identifies an ambiguity in its work and remedies it in [13]. A timed semantics has been studied which supports the analysis of properties in time related orchestration patterns [15]. The equivalence relation based on the timed semantics is also studied. Based on these deep discussions on its operational semantics, the model checking technology can be used to verify the correctness of Orc programs with respect to formal specifications. The researches of M. AlTurki are good examples to this direction [2,3]. In the denotational semantics aspect, C.A.R. Hoare et al. has proposed a tree semantics for Orc language and also proved the key algebra laws in this model [8]. However, the trace equivalence is not convenient in verifying Orc programs with respect to specifications which subjects to a refinement relation.

This paper proposes a denotational semantics for Orc language using UTP methods [7]. It shows that a orchestration language as Orc can be formalized to a state based logical system using the UTP method. An Orc program is formalized by a predicate with structured traces and program status. Simple predicate calculus is sufficient to prove the laws of Orc combinators directly. A refinement relation is proposed to provide a partial order over the domain of Orc programs according to the nondeterminism. Based on the semantical model and the refinement relation, one can determine whether an Orc program satisfies a given specification, which guarantees the correctness of Orc programs. The conclusion can be checked by theorem provers such as Isabelle and PVS. This is the direction we are willing to take and this paper can be our first step.

The reminder of the paper is organized as follows. Section 2 introduces the Orc language briefly. One can learn how an Orc program looks like and which orchestration patterns it can implement. Section 3 proposes the denotational semantics for Orc. Every Orc program corresponds to a predicate satisfying the healthiness conditions. Section 4 discusses the equivalence relation and proves some properties of specific Orc programs. Finally, section 5 concludes the whole paper and mentions some future works.

2 Orc Language

In this paper, we only consider the core Orc concurrency calculus which includes primary site calls and four combinators. The Syntax of Orc language is proposed as follows:

$$\begin{aligned} Formal &::= x \mid y \mid z \mid ... \\ Actual &::= a \mid b \mid c \mid ... \\ p &\in \quad Formal \cup Actual \\ P, Q &::= \mathbf{0} \mid M(p) \mid P || Q \mid P > x > Q \mid P < x < Q \mid P |> Q \end{aligned}$$

where the notation p represents a parameter. The parameter in a definition can be either formal parameter or actual parameter.

0 is a program which contains no event and terminates immediately.

$M(p)$ invokes a site call with the site name M using the parameter list p. The program publishes the response from the site M. A site call can only be invoked when every formal parameter has a value. If the result of the site is more than one, the program publishes them in a tuple.

Site calls together with **0** are considered as primitives in Orc language.

The parallel combinator $P||Q$ lets two programs execute in parallel. The results of the two programs will be published in arbitrary order. The invocations performed by P and Q are independent. If P and Q invoke the same site, the program will perform two invocations.

The sequential combinator $P > x > Q$ first executes the program P. When P publishes a result, Q assigns it to the variable x and begin the execution in parallel with the rest part of P. Note that every output of P will enable a new execution of Q with it.

The pruning combinator $P < x < Q$ is firstly named as asymmetric parallel combinator. P and Q are executed in parallel except that the parameter x in P should take the value from the first result published by Q. Q will be forced to halt just after publishing a result.

The otherwise combinator $P| > Q$ was first introduced with the notation ; in Orc programming language. It first executes P. If P halts without any output, Q will be executed. Otherwise, Q will never start. This combinator can support the failure handling and is essential for dealing with web environment.

The Orc language can orchestrate the web services to complete user requirements. The three combinators have the capability to construct sufficient patterns needed in the web applications on high abstract level. For example, consider the following orchestration problems. Linda plans to have a weekend in Shanghai. She has to book an airplane ticket and a local hotel in Shanghai. She'd like the airline of either the Air China(CA) or American Air(AA). She also wants to book a room of Hilton Hotel in Shanghai. She prefers a suite to a guest room. The trip will happen when both the flight and the hotel are reserved. The following Orc program can make her plan work and send a email to her when the reservation is done.

$(resv(x, y) < x < (CA(l)||AA(l)) < y < (H(s)| > H(g)) > z > email(Linda, z)$

Where $CA(l)$ and $AA(l)$ is two independent site calls requiring the airline l to the Air China site CA and the American Air site AA. The pruning combinator ensures that either's returning is acceptable and is passed to the reservation engine $resv$. $H(s)$ requires a suite and $H(g)$ requires a guest room of Hilton Hotel. The otherwise combinator makes $H(s)$ has higher priority than $H(g)$. The guest room is involved only if it fails to obtain a suite. The reservation engine $resv$ makes the reservations done as soon as it gets both the flight info and hotel info and sends the final result to Linda through email.

3 Denotational Semantical Model for Orc

3.1 Alphabet

The denotational semantics of a program in Orc language can be specified as a predicate with the following alphabet.

$$\alpha = \{tr, st\}$$

tr is an interaction trace observed from the execution of the program. st reflects the execution status of the program. We say a program can perform a trace tr with a status st when the pair (tr, st) satisfies its semantics.

The elements of the trace belongs to the following event set

$$\Sigma = \{k.\hat{x}?u, k!v, k!\theta \mid k \in N \wedge x \in F \wedge u, v \in C\}$$

where N is a set of instance names. F is the set of formal variables which have no actual values. C is the set of actual values. It is required that every instance of a site has a distinct name. We use $N(e)$ to get the name of an event e. The notation $\hat{x}$ represents a label corresponding to the variable x. We use the notation $L(e)$ to retrieve the label from a input event e. If $e = k.\hat{x}?u$, then $L(e) = \{x\}$. The notation ? means this event is an input event and the notation ! represents an output event. The part after ? or ! is an expression. In the semantical definition, an expression can contain free variables which is waiting to be assigned with actual values. The occurrences of a variable in an expression is called its free occurrences. We define the notation $V(e)$ to denote the free variables in the event e. For example, if $e = k!3x$, then $V(e) = \{x\}$. Note that in the tr observed from the program's behavior, every free variable has an actual value which is in set C. There is a special value denoted as θ denoting a signal which passes the control to the program receiving it. We use the notation $\Sigma!$ and $\Sigma?$ to denote the subset of output events and input events respectively. Moreover, we extend the above functions over traces. $N(tr) =_{df} \cup_{a \in tr} N(a)$, $L(tr) =_{df} \cup_{a \in tr} L(a)$, $V(tr) =_{df} \cup_{a \in tr} V(a)$.

The status variable st which can yield value in the set $ST = \{run, comp, halt, div\}$ reflects the execution status of the program. $st = run$ represents the program is doing its work with given inputs. $st = comp$ means the program finishes its work successfully and returns a result. $st = halt$ represent the program terminates without a result. And finally $st = div$ means the program enters the divergent status in which it only engages internal communications and never performs an external communication.

Let $st_1, st_2 \in ST$ be status variables. We define a binary composition $|$ as follows:

$$st_1|st_2 =_{df} \begin{cases} div, & \text{if } st_1 = div \vee st_2 = div \\ comp, & \text{if } st_1 = st_2 = comp \\ run, & \text{if } (st_1 = run \wedge st_2 \neq div) \vee (st_1 \neq div \wedge st_2 = run) \\ halt, & \text{Otherwise} \end{cases}$$

We can obtain the result status after the composition. Some properties of the composition can be observed from the definition. For example, if a component is divergent, the composite program is divergent. The composite program is complete if both of the components are complete. We can also find that the status composition is commutative and associative. So we can define a unified operator.

$|_{i \in [1,n]} st_i =_{df} st_1|st_2|...|st_n$

We can use Table 1 to illustrate the function of the status composition $|$.

3.2 Predicate Semantics

In our denotational semantical model, every site call in Orc language can be specified as a predicate. All the combinators can be formalized by predicate expressions. Therefore every expression in Orc program can be formalized by a set of free variables and a predicate over the alphabet $\{tr, st\}$.

Table 1. Definition of operator $|$

$st_1 \| st_2$	div	halt	comp	run
run	div	run	run	run
comp	div	halt	comp	
halt	div	halt		
div	div			

Before discussing the combinators in Orc language, we first define several operators in order to simplify the expression.

Definition 1 (Successor). *Let P be an Orc program and D be a predicate over the alphabet $\{tr, st, tr', st'\}$. The successor operator $P; D$ does some modifications to the result of the predicate P.*
$P(tr, st); D(tr, st, tr', st') =_{df} (\exists m, n \bullet P(m, n) \wedge D(m, n, tr', st'))[tr/tr', st/st']$

Note that the sequential composition we defined here is left-associative. When we write $P; Q; R$, we mean $(P; Q); R$.

Definition 2 (Nondeterministic Choice). *Let P, Q be Orc programs. The nondeterministic choice $P \sqcap Q$ can perform either like P or like Q.*

$$P \sqcap Q =_{df} P \vee Q$$

Definition 3 (Conditional Choice). *Let P, Q be Orc programs and b be a boolean expression. The conditional choice $P \lhd b \rhd Q$ behaves like P if b holds and behaves like Q otherwise.*

$$P \lhd b \rhd Q =_{df} (b \wedge P) \vee (\neg b \wedge Q)$$

Definition 4 (After). *Let P be an Orc program and s be a trace. The behavior of the program P/s is obtained by removing prefix s from the behaviors of program P which contain this prefix.*

$$P/s =_{df} P[(s \cdot tr)/tr]$$

Definition 5 (Hiding). *Let P be an Orc program and X be a set of variable names. The hiding operator hides the events which have the label within the set X from its behavior.*
$P \backslash X =_{df} P; (tr' = tr \backslash X \wedge st' = div \lhd Div(tr, X) \rhd tr' = tr \backslash X \wedge st' = st)$
where $Div(tr, X) =_{df} \forall n \bullet length(tr \uparrow X) > n$

$$tr \backslash X =_{df} \begin{cases} \epsilon, & \textit{if } tr = \epsilon \\ tail(tr) \backslash X, & \textit{if } L(head(tr)) \cap X \neq \emptyset \\ \langle head(tr) \rangle \cdot (tail(tr) \backslash X), & \textit{Otherwise} \end{cases}$$

$$tr \uparrow X =_{df} \begin{cases} \epsilon, & \textit{if } tr = \epsilon \\ \langle head(tr) \rangle \cdot (tail(tr) \uparrow X), & \textit{if } (V(head(tr)) \cup L(head(tr))) \cap X \neq \emptyset \\ tail(tr) \uparrow X, & \textit{Otherwise} \end{cases}$$

Note that the hiding operator can generate divergence. If a program performs unbounded communications through the hiding variables in X, we claim the program diverge since the hiding.

3.3 Healthiness Conditions

In our principle, every Orc program can be specified by a special predicate. However, some predicates are not appropriate for specifying Orc programs. In this subsection, we will propose some restriction which we called healthiness conditions to indicate which kind of predicates can be used to describe the behaviors of Orc programs.

At first, we define an partial order applying on the predicate space.

Definition 6 (Refinement Relation). *Let P, Q be predicates applying on the same alphabet. We say P is a refinement of Q, denoted as $P \sqsupseteq Q$, iff $[P \Rightarrow Q]$. The square brackets apply universal quantifier on every free variable in the formula.*

There are some healthiness conditions we should follow in the Orc Language. The healthiness conditions can help us to address which kind of predicate is good enough to become specifications of Orc programs. With the healthiness conditions, some essential properties of the Orc programs can be guaranteed.

In Orc language, every primitive generates an instance of a site working with the input parameters and returns at most one output. According to this requirement, we can figure out that an Orc program which is composed by these primary expressions should only performs particular traces which we call valid traces.

Definition 7 (Valid Trace). *A trace tr is called valid, represented as $valid(tr)$, if it satisfies the following condition.*

$$valid(tr) =_{df} tr = \epsilon \vee \begin{pmatrix} tr \neq \epsilon \wedge \forall k \in N(tr) \bullet \\ tr_{\{k\}} \in \{s \cdot t \mid s \in \Sigma?^* \wedge t \in \Sigma!^\epsilon\} \end{pmatrix}$$

where for any set of instance names K,

$$tr_K =_{df} \begin{cases} \epsilon, & \textit{if } tr = \epsilon \\ \langle head(tr)\rangle \cdot (tail(tr)_K), & \textit{if } N(head(tr)) \cap K \neq \emptyset \\ tail(tr)_K, & \textit{Otherwise} \end{cases}$$

The notation $\Sigma!^\epsilon = \Sigma! \cup \{\epsilon\}$. The valid property means one instance of site calls has zero or more input and zero or one output; any input should take place before its corresponding output.

H: Invalid traces can be only observed in a divergent program which is unexpected.

$$[P = valid(tr) \Rightarrow P]$$

We can define a mapping which can convert a predicate to satisfy the corresponding healthiness condition. We call this kind of mappings as healthiness mappings.

$$\mathbf{H}(P) =_{df} valid(tr) \Rightarrow P$$

Any predicate allowing an invalid trace will be mapped to the predicate $True$. We call it chaos which is most nondeterministic.

The Orc language is designed to specify the cooperation relations between programs in web environment, the events observed in the behaviors indicate the asynchronous interactions between web applications. The order of event in the sending side is often observed to be different from the order in the receiving side. According to this fact, we first define an equivalence relation over the valid traces.

Definition 8 (Permutable Equivalence). *Two valid traces tr_1, tr_2 are said to be permutable equivalent, denoted as $tr_1 \approx tr_2$, iff tr_1 is a permutation of tr_2 and vice versa.*

We define a mapping $PE(tr)$ to get all the traces satisfying the permutable equivalence relation with trace tr. Formally,

$$PE(tr) =_{df} \begin{cases} \{s \mid tr \approx s\}, & \text{if } valid(tr) \\ \emptyset & \text{if } \neg valid(tr) \end{cases}$$

Orc1: If a trace can be performed by a program, all traces permutable equivalent to it can also be performed by this program.

$$[\exists s, t \bullet s \approx t \Rightarrow (P[s/tr] \Leftrightarrow P[t/tr])]$$
$$\mathbf{O1}(P) =_{df} P; Perm$$

where $Perm =_{df} (tr' \in PE(tr) \wedge st' = st)$

Orc2: We consider the divergence as an unexpected and uncontrollable behavior of a program. If a program diverges, its further behavior is considered as a chaos.

$$[P[div/st] \Rightarrow \forall s \bullet P[(tr \cdot s)/tr]]$$
$$\mathbf{O2}(P) =_{df} P; Filter$$

where $Filter =_{df} tr \preceq tr' \lhd st = div \rhd II$
$II =_{df} tr' = tr \wedge st' = st$

Orc3: The traces performed by every program yield the prefix-closure property. If a trace can be observed from the behavior of a program, all its prefixes should also be observed.

$$[\exists s, t \bullet P[s \cdot t/tr] \Rightarrow P[s/tr, run/st]]$$
$$\mathbf{O3}(P) =_{df} P; Pref$$

where $Pref =_{df} (tr' \preceq tr \wedge st' = run)$

In summary, we obtain a mapping $O =_{df} (O3 \circ O2 \circ O1) \circ H$ which combines all the healthiness mappings.

3.4 Semantics of Primitives

We can propose the denotational semantics of the primitives in Orc language.

0 is a program which terminates immediately without performing any event.

$$\mathbf{0} =_{df} tr = \epsilon \wedge (st = run \vee st = comp)$$

Obviously it satisfies the healthiness conditions and is a healthy Orc program.

A site call $M(p)$ invokes the site M with a parameter p. p can be either an actual or formal parameter list. However, the signal θ can be never used as a parameter in the syntax. Every execution of a site call engages an instance which has a distinct name k.

Therefore, we can propose a common pattern of the site call. Every site call has a definition following this pattern.

$$M(p) =_{df} O\begin{pmatrix} tr = \langle k.\hat{x}?p\rangle & \wedge\, st = halt \\ \vee\, tr = \langle k.\hat{x}?p, k!f_M(p)\rangle & \wedge\, st = comp \end{pmatrix}$$

The function $f_M(p)$ represents the relation between the input parameter and the output result. From the common definition of the site call we can find all the properties we mentioned before. When the parameter is ready, the program will accept the invocation and stay in the run status until the output result returns. This behavior is not written explicitly because it is included by the healthiness mapping O. The behavior

$st = halt$ can be observed if there is a failure taking place and the result never comes. The behavior where $st = comp$ is performed when the result returns.

$let(p)$ publishes the value of the parameter immediately. In that circumstance, $let(p)$ is a local computation not a site call. But if we neglect the differences between local computation resource and the remote services, we can consider it as a special site call which returns the value of the parameter without suffering any network failure. Let $\mathrm{fn}(let) = \{x\}$, then we define

$$let(p) =_{df} O(tr = \langle k.\hat{x}?p, k!p\rangle \wedge st = comp)$$

For example, the behavior of the actual site call $let(3)$ with $\mathrm{fn}(let) = \{x\}$ is as follows. Note that $V(let(x)) = \{x\}$ but $V(let(3)) = \emptyset$.

$$let(3) =_{df} O(tr = \langle k.\hat{x}?3, k!3\rangle \wedge st = comp)$$

There is a special site call which always fails to obtain the result. We call it $sink$, which can be defined as follows. It never reaches the $comp$ status.

$$sink(p) =_{df} O(tr = \langle k.\hat{x}?p\rangle \wedge st = halt)$$

A conditional choice if is a site call to valuate a boolean expression. When the boolean expression is true, then a signal is sent out, otherwise, no response will be provided.

$$if(p) =_{df} O\begin{pmatrix} p \wedge tr = \langle k.\hat{x}?p, k!\theta\rangle \wedge st = comp \\ \vee \neg p \wedge tr = \langle k.\hat{x}?p\rangle \quad \wedge st = halt \end{pmatrix}$$

The parameter of if should only be a boolean expression.

There are two additional programs in Orc program language which are named as $signal$ and $stop$. They need no input parameter and have definite behaviors. $signal$ terminates with publishing a signal θ. $stop$ terminates without any response. They are not ordinary site calls but also yield to the healthiness conditions.

$$signal =_{df} O(tr = \langle k!\theta\rangle \wedge st = comp)$$
$$stop =_{df} O(tr = \epsilon \wedge st = halt)$$

From the definition we can find that $signal$ has similar behavior as the program $if(true)\backslash\{x\}$ while $stop$ is behavioral equivalent to $if(false)\backslash\{x\}$.

3.5 Semantics of Combinators

Let P and Q be Orc programs with $N(P) \cap N(Q) = \emptyset$. Then for any combinator in Orc language, we have $\mathrm{fn}(P \text{ op } Q) = \mathrm{fn}(P) \cup \mathrm{fn}(Q)$ and $N(P \text{ op } Q) = N(P) \uplus N(Q)$.

The definition of the parallel combinator is as follows.

Definition 9 (Parallel Combinator)
$P||Q =_{df} P||_{SM}Q$
$P||_{SM}Q =_{df} \exists 1.tr, 1.st, 2.tr, 2.st \bullet (1.P \wedge 2.Q); SM; Filter$
$SM =_{df} tr' \in (1.tr|||2.tr) \wedge st' = 1.st|2.st$
where the binary operator $|||$ *represents the interleaving composition of two traces.*

The parallel combinator can be specified as the interleaving of the behaviors of the two participants. Note that the behaviors of the two participants never interfere each other. So in the definition, we use the notations $1.tr, 2.tr$ to substitute the original tr in the two participants respectively. Then the subsequent predicate SM applying on the alphabet $\{1.tr, 2.tr, 1.st, 2.st, tr, st, tr', st'\}$ merges the behaviors performed by the two participants and forms the behavior of the parallel program.

The definition of the sequential combinator is as follows.

Definition 10 (Sequential Combinator)

$$P > x > Q =_{df} (P \wedge tr\uparrow ! = \epsilon) \vee \left(\begin{array}{l} \exists tr_0, st_0 \bullet tr_0 \preceq tr \wedge tr_0\uparrow ! = \epsilon \wedge P(tr_0 \cdot \langle k!a \rangle, st_0) \\ \wedge\ ((P/tr_0 \cdot \langle k!a \rangle) > x > Q \ || \ Q[tr[a/x]/tr])[(tr - tr_0)/tr]\backslash\{x\} \end{array} \right)$$

where $(s \cdot t) - s = t$. *The notation* $tr\uparrow !$ *is short for* $tr \uparrow \Sigma!$.

In the definition we hide the intermediate communications which including the output events of P and the input events of Q labeled by x, which consists the value passing from P to every instance of Q. The value passing restricts the free variable x in the subsequent program to the value published by the previous one. Every output of P activates an independent instance of Q initialized by the output value. We consider θ has no semantical effect in the substitution. In other terms, $Q[tr[\theta/x]/tr]$ is the same as Q itself. It is only a signal to activate the subsequent program without passing any value. If the variable x in combinator is not a free variable of Q, Q can be activated but can never get any progress. We can abbreviate the combinator as $\gg$ if we do not need to restrict the value passing to particular labels.

The definition of the pruning combinator is as follows.

Definition 11 (Pruning Combinator)

$$P < x < Q =_{df} P||_{AM_x} Q$$

$$P||_{AM_x} Q =_{df} \exists 1.tr, 1.st, 2.tr, 2.st \bullet (1.P \wedge 2.Q); AM_x; Filter$$

$$AM_x =_{df} \left(\begin{array}{l} 2.tr\uparrow ! = \epsilon \wedge \\ \left(\begin{array}{l} x \notin V(1.tr) \wedge tr' \in 1.tr|||2.tr \wedge st' = 1.st|2.st \\ \vee\ \exists s, t \bullet 1.tr = s \cdot t \wedge x \notin V(s) \wedge x \in V(head(t)) \wedge \\ \quad \exists r \bullet r \in (s|||2.tr) \wedge tr' = (r \cdot t)\backslash\{x\} \wedge st' = 1.st|2.st \end{array} \right) \\ \vee\ \exists s_2, t_2 \bullet 2.tr = s_2 \cdot \langle k!a \rangle \cdot t_2 \wedge s_2\uparrow ! = \epsilon \wedge \\ \left(\begin{array}{l} x \notin V(1.tr) \wedge tr' \in (1.tr|||s_2) \wedge st' = 1.st \\ \vee\ \exists s_1, t_1 \bullet 1.tr = s_1 \cdot t_1 \wedge x \notin V(s_1) \wedge x \in V(head(t_1)) \wedge \\ \quad \exists q \in (s_1|||s_2) \bullet tr' = q \cdot (t_1[a/x]\backslash\{x\}) \wedge st' = 1.st \end{array} \right) \end{array} \right)$$

The pruning combinator is formerly called asymmetric parallel composition. The definition of the combinator has the similar form as parallel combinator. But its merge part is more complicated. The x in the combinator is also a label indicating where the value passing takes place. If the program P in $P < x < Q$ has a free variable x, its execution will be blocked when it needs the value of x which is expected to be provided by the first output from Q. Once Q publishes its first output, it will be forced to terminate. Hence, the behavior of $P < x < Q$ contains the behavior of Q only before its first output. Note that even if Q diverges afterwards, it never causes the divergence of the composed program. Similar to the sequential combinator, the pruning combinator can be abbreviated as $\ll$ if the restriction of the value passing is unnecessary.

The otherwise combinator can be defined as follows.

Definition 12 (Otherwise Combinator)

$$\begin{array}{ll} P| > Q =_{df} & (P \wedge \neg(st = halt \wedge tr\uparrow ! = \epsilon)) \vee \\ & (\exists s \bullet P[s/tr, halt/st] \wedge s\uparrow ! = \epsilon \wedge Q) \end{array}$$

From the definition we can find that Q can only be executed when P end up with $halt$ status and has no output. In this case, $P| > Q$ behaves like Q.

Considering the combinators we defined, we can prove that they preserve the healthiness conditions.

Theorem 1. *If P and Q satisfy all the healthiness conditions, then so is P op Q where op $\in \{||, \gg, \ll, |>\}$. Concretely,*
(1.1) $O(P)||O(Q) = O(P||Q)$
(1.2) $O(P) > x > O(Q) = O(P > x > Q)$
(1.3) $O(P) < x < O(Q) = O(P < x < Q)$
(1.4) $O(P)|> O(Q) = O(P|> Q)$

Moreover, we can prove that the mapping O is monotonic with respect to the refinement relation.

Theorem 2. *Let P and Q be predicates. If $P \sqsupseteq Q$ then $O(P) \sqsupseteq O(Q)$.*

According to the Taski's Fixpoint Theory, we can claim that the fixpoint of the mapping O does exist, which leads us to the following proposition.

Theorem 3. *A Orc program can be expressed by a predicate which is a fixpoint of the mapping O. And the set of Orc programs forms a complete lattice $(\mathbb{O}, \sqsupseteq)$.*

Additionally, we can find the top element and the bottom element of the complete lattice.

$$\top_{\mathbb{O}} = O(False) = \neg valid \qquad \text{Since } \; P \sqcap \neg valid = P$$

$$\bot_{\mathbb{O}} = O(True) = True \qquad \text{Since } \; P \sqcap True = True$$

The bottom element $\bot_{\mathbb{O}}$ is also the bottom of the predicate domain. It can perform any trace with any arbitrary status, which is what we called chaos. We will omit the suffix $\mathbb{O}$ in the following discussion.

4 Equivalences and Properties

The Orc language is based on the Kleene Algebra which contains several structural axioms. Based on the denotational semantics proposed in this paper, we can prove all these properties directly using the predicate calculus which can be verified by theorem provers. Moreover, our denotational model can specify the divergent behaviors of an Orc program and provide a refinement ordering according to the nondeterminism.

In our model, the denotational semantics uses predicate to specify the set of behaviors so that one program is mapped to a predicate over the alphabet $\{tr, st\}$. Most of the properties we want to prove below are equations. In the semantical level, this corresponds to predicate equivalence.

Definition 13 (Strong Equivalence). *Let P and Q be two predicates. we say P and Q are strong equivalent, denoted as $P \equiv Q$, if and only if $P \sqsupseteq Q$ and $Q \sqsupseteq P$.*

There is another refinement relation which we call weak refinement. This equivalence does not consider the instance names in the trace. The relation is reasonable because it is consistent with the view of users. When a user executes a program, he is not concerned with the information which instance of the site performs the computation. It is sufficient for him to know the result published by the whole computation.

Definition 14 (Weak Refinement). *Let P, Q be predicates and a set of instance name N. We say P is a refinement of Q applying on the alphabet $\{tr, st\}$, denoted as $P \geq Q$, if and only if $fn(P) = fn(Q)$ and $\forall tr, st \bullet (P \downarrow N) \Rightarrow (Q \downarrow N)$.*

where $P \downarrow N =_{df} P; (tr' = tr \downarrow N \wedge st' = st)$
and $tr \downarrow N$ hides all the instance names in the events of trace tr.

With the weak refinement relation, we can define weak equivalence.

Definition 15 (Weak Equivalence). *Let P and Q be two predicates. we say P and Q are weak equivalent, denoted as $P = Q$, if and only if $P \geq Q$ and $Q \geq P$.*

In the following, when we mention the term "equivalence", we refer to the weak equivalence.

Let P, Q, R be Orc programs. We can prove properties of Orc programs using predicate calculus.

The parallel combinator has the following properties.

Properties ($||$):
($||$-1) $P||Q \equiv Q||P$
($||$-2) $(P||Q)||R \equiv P||(Q||R)$
($||$-3) $P||\mathbf{0} \equiv P$

Proof: Note that the operator $|||$ and $|$ are both commutative and associative. Hence ($||$-1) and ($||$-2) can be obtained by predicate calculus. Moreover, since ϵ is the unit element of the operator $|||$, $comp$ are the units of the operator $|$, the properties of ($||$-3) can be obtained directly. □

The sequential combinator has the following properties.

Properties ($\gg$):
($\gg$-1) $\mathbf{0} \gg P \equiv \mathbf{0}$
($\gg$-2) $P \gg Q \equiv P$ if $P\uparrow? \equiv P$ where $P\uparrow? =_{df} P; (tr' = tr\uparrow? \wedge st' = st)$
($\gg$-3) $P \gg \mathbf{0} \equiv P\uparrow?$
($\gg$-4) $(P||Q) \gg R \equiv (P \gg R)||(Q \gg R)$
($\gg$-5) $(P > x > Q) > y > R \equiv P > x > (Q > y > R)$, if $x \notin$ fn(R)
($\gg$-6) $P > x > let(x) = P$

Proof:

$$\begin{array}{lll}
(\gg\text{-}1)\ \mathbf{0} \gg P & \equiv (\mathbf{0} \wedge tr\uparrow! = \epsilon) \vee False & \{\text{def } \mathbf{0}; \text{predicate calc}\} \\
& \equiv \mathbf{0} & \{\text{def } \mathbf{0}\} \\
(\gg\text{-}2)\ P \gg Q & \equiv P\uparrow? \gg Q & \{P \equiv P\uparrow?\} \\
& \equiv (P\uparrow? \wedge tr\uparrow! = \epsilon) \vee False & \{\text{predicate calc}\} \\
& \equiv P & \{\text{predicate calc}; P\uparrow? \equiv P\}
\end{array}$$
□

The properties of the pruning combinator is as follows.

Properties ($\ll$):
($\ll$-1) $P < x < \mathbf{0} \equiv P\backslash\{x\}$
where $P\backslash\{x\} =_{df} P; (tr' = tr\backslash\{x\} \wedge st' = st)$
($\ll$-2) $(P||Q) < x < R \equiv (P < x < R)||Q$, if $x \notin$ fn(Q)

($\ll$-3) $(P>y>Q)<x<R \equiv (P<x<R)>y>Q$, if $x \notin$ fn(Q)
($\ll$-4) $(P<x<Q)<y<R \equiv (P<y<R)<x<Q$, if $y \notin$ fn(Q) and $x \notin$ fn(R)
($\ll$-5) $\mathbf{0} \ll M(c) \equiv M(c) \gg \mathbf{0}$ where $c \in Actual$
($\ll$-6) $P \ll M(c) = P||(M(c) \gg \mathbf{0})$ where $c \in Actual$

Proof:
($\ll$-1) $P<x<\mathbf{0} \equiv P||_{AM_x}\mathbf{0}$

$$\begin{aligned}
&\equiv \exists 1.tr, 1.st, 2.tr, 2.st \bullet (1.P \wedge 2.tr = \epsilon \wedge 2.st = comp); AM_x; Filter\\
&\equiv \exists 1.tr, 1.st \bullet 1.P \wedge\\
&\quad \begin{pmatrix} x \notin V(1.tr) \wedge tr' \in 1.tr \wedge st' = 1.st \\ \vee\ \exists s, t \bullet 1.tr = s \cdot t \wedge x \notin V(s) \wedge x \in V(head(t)) \wedge \\ tr' = (s \cdot t)\backslash\{x\} \wedge st' = 1.st \end{pmatrix}; Filter\\
&\equiv \exists 1.tr, 1.st \bullet 1.P \wedge (tr' = 1.tr\backslash\{x\} \wedge st' = 1.st); Filter\\
&\equiv P; (tr' = tr\backslash\{x\} \wedge st' = st); Filter\\
&\equiv P\backslash\{x\}
\end{aligned}$$

The properties of the otherwise combinator is as follows.

Properties ($|>$):
($|>$-1) $\mathbf{0}|> P \equiv \mathbf{0}$
($|>$-2) $stop|> P \equiv P$
($|>$-3) $(P|> Q)|> R \equiv P|> (Q|> R)$

Proof:

$$\begin{aligned}
(|>\text{-1})\ \mathbf{0}|> P &\equiv (\mathbf{0} \wedge \neg(st = halt \wedge tr\uparrow! = \epsilon))\vee\\
&\quad (\exists s \bullet \mathbf{0}[s/tr, halt/st] \wedge s\uparrow! = \epsilon \wedge P) && \{\text{def } |>\}\\
&\equiv \mathbf{0} \vee False && \{\neg\mathbf{0}[halt/st]\}\\
&\equiv \mathbf{0}\\
(|>\text{-2})\ stop|> P &\equiv O(tr = \epsilon \wedge st = halt)|> P && \{\text{def } stop\}\\
&\equiv O(False \vee P) && \{\text{def } |>\}\\
&\equiv P && \{O(P) \equiv P\}
\end{aligned}$$

□

The nondeterminism in Orc language is mainly introduced by the arbitrary ordering of the events in parallel composition. In fact, the nondeterminism is concealed in the semantics of primitives. For example, consider a remote site *copy* which returns the value of the input parameter. There may be a network failure during its execution and the result never returns. Hence its behavior is equivalent to the following one.

$$copy(p) =_{df} let(p) \sqcap sink(p)$$

From this view, we can obtain that the behavior of *copy* may succeed just as the local computation let and may also halt like the failure site $sink$. Actually, almost every remote site call has the possibility to halt with a failure. This kind of nondeterminism is inherent in web environment. And apparently we have $let(p) \geq copy(p)$ according to our weak refinement relation.

The bottom element in Orc domain has the most nondeterminism, which makes us call it chaos. It is the zero element of parallel combinator $||$ and the left-zero of pruning combinator $\ll$.

Properties ($\bot$):
($\bot$-1) $\bot||P \equiv \bot$

($\perp$-2) $\perp \ll P \equiv \perp$
($\perp$-3) $\perp | > P \equiv \perp$
($\perp$-4) For any Orc program P, $P \sqsupseteq \perp$

Note that $\perp \gg P \equiv \perp$ does not hold when P has no output. For example, $\perp \gg 0 \not\equiv \perp$.

The behavior of $\perp$ can never be simulated by other Orc programs. But there exists some programs which diverge after being invoked. For example, consider the following recursive program.

$$Rec(x) =_{df} let(x) > x > Rec(x)$$

In this definition, all the outputs generated by the invocations of $let(x)$ are consumed by their successor and the program never terminates. The observer cannot observe any event after the invocation, but the program never stops. This kind of divergence is also called as a livelock.

We can verify that the behavior of $Rec(x)$ equals to the following program.

$$Rec(x) = O(tr = \langle k.\hat{x}?x\rangle \wedge st = div)$$

Note that not all programs which has a component as Rec always lead to divergence. See the property ($\gg$-3) above ($\mathbf{0} \gg Rec \equiv \mathbf{0}$), if the left component terminates with no output, the program never diverges because its behavior is equivalent to the left component.

In Orc language, a program *halts* if it ends up with $halt$ status and has no output. The otherwise combinator is introduced to handle the halt program. Note that the halt behavior we defined above is a different issue. In common perspective, the halt program is unexpected. The program has less probability to halt is considered better. We define the following filter operation.

$$P \downarrow halt =_{df} P; (\perp' \lhd st = halt \wedge tr\uparrow ! = \epsilon \rhd II)$$

The filter considers every halt execution of the program as a divergence. This view wipes out the halt executions of programs and generate a specific ordering to reflect the above perspective.

Definition 16 (Halt Ordering). *Let P and Q be Orc programs. we say P is the refinement of Q with respect to halt, denoted as $P \rtimes Q$, if and only if $P \downarrow halt \sqsupseteq Q \downarrow halt$.*

Definition 17 (Weak Halt Ordering). *Let P and Q be Orc programs. we say P is weak refinement of Q with respect to halt, denoted as $P \gtrdot Q$, if and only if $P \downarrow halt \geq Q \downarrow halt$.*

With this ordering, we can determine that $let(p) \gtrdot sink(p)$ while we cannot tell $let(p) \geq sink(p)$. Moreover, we can determine the ordering between the following two programs. There is a program called $cord(p)$ which coordinates a site call $copy(p)$ and the unfail local publish $let(p)$. We combine them with the otherwise combinator $cord(p) =_{df} copy(p)| > let(p)$. According to definition of otherwise combinator, the program first launches the site call. If it fails, a local publication is made. Then we can find out

$$cord(p) \gtrdot copy(p)$$

This result supports the perspective that the otherwise combinator improves the behavior of programs.

5 Conclusion and Future Work

This paper proposes a denotational semantics for Orc language. We give precise definition for the site calls and Orc combinators. Through these definitions, every Orc program is formalized by a predicate yielding healthiness conditions. Based on the semantical model, the algebra laws of the Orc concurrency calculus can be proved by predicate calculus. The model can express both the specification and the implementation of a system. With the help of the refinement principle, we can figure out whether a program satisfies a specification as well as compare the behaviors of two programs according to their nondeterminism. Our semantical model transforms the verification process to the predicate computation which can be assisted by theorem provers such as Isabelle and PVS.

In our future work, we will extend our model by adding the refusals to the semantical domain. A program may diverge after it publishes some result. This kind of programs appears to be better than Rec. For example, consider the following program:

$$Pub(x) =_{df} let(x)||Rec(x)$$

This program can publish an output before it reaches a livelock. However, in our model we have $Pub(x) \equiv Rec(x)$. We cannot tell the difference between this two programs in our current model. This problem can be solved by considering the refusals in each step of the program execution.

In addition, we will discuss the soundness and the completeness of the model with respect to the standard operational semantics. The UTP method leads us an easier way to establish a link between these two semantics. We can combine the service specification model proposed in [6] to obtain a complete view of a service oriented system. It makes the verification more reliable in dealing with practical applications.

Acknowledgement. This research was partly supported by National Basic Research Program of China (No. 2005CB321904), National High Technology Research and Development Program of China (No. 2007AA010302), National Natural Science Foundation of China (No. 90718004) and PhD Program Scholarship Fund of ECNU (No. 2010046).

References

1. Alonso, G., Kuno, H., Casati, F., Machiraju, V.: Web Services: Concepts, Architectures and Applications. Springer, Heidelberg (2003)
2. AlTurki, M., Meseguer, J.: Real-time rewriting semantics of orc. In: Proc. PPDP 2007: 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 2007, pp. 131–142. ACM, New York (2007)
3. AlTurki, M., Meseguer, J.: Reduction semantics and formal analysis of orc programs. Electronic Notes in Theoretical Computer Science 200(3), 25–41 (2008)
4. Cook, W.R., Patwardhan, S., Misra, J.: Workflow patterns in orc. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 82–96. Springer, Heidelberg (2006)
5. Erl, T.: Service-Oriented Architecture (SOA): Concepts, Technology, and Design. Prentice Hall PTR, Englewood Cliffs (2005)
6. He, J.: Service refinement. Science in China Series F: Information Sciences 51(6), 661–682 (2008)

7. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice Hall International Series in Computer Science (1998)
8. Hoare, T.: A tree semantics of an orchestration language. In: Proc. NATO Advanced Study Institute, Engineering Theories of Software Intensive Systems. NATO ASI Series (2004)
9. Kitchin, D., Cook, W.R., Misra, J.: A language for task orchestration and its semantic properties. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 477–491. Springer, Heidelberg (2006)
10. Kitchin, D., Quark, A., Cook, W.R., Misra, J.: The orc programming language. In: Lee, D., Lopes, A., Poetzsch-Heffter, A. (eds.) FMOODS 2009. LNCS, vol. 5522, pp. 1–25. Springer, Heidelberg (2009)
11. Misra, J., Cook, W.R.: Computation orchestration. Software and System Modeling 6(1), 83–110 (2007)
12. Orc Language Project. Orc Program Language Demo, `http://orc.csres.utexas.edu/tryorc.shtml`
13. Vardoulakis, D., Wand, M.: A compositional trace semantics for orc. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 331–346. Springer, Heidelberg (2008)
14. Wehrman, I., Kitchin, D., Cook, W.R., Misra, J.: Properties of the timed operational and denotational semantics of orc. Technical report, Department of Computer Science, The University of Texas, Austin (December 2007)
15. Wehrman, I., Kitchin, D., Cook, W.R., Misra, J.: A timed semantics of orc. Theoretical Computer Science 402(2-3), 234–248 (2008)

An Extended cCSP with Stable Failures Semantics

Zhenbang Chen[1] and Zhiming Liu[2]

[1] National Laboratory for Parallel and Distributed Processing, Changsha, China
[2] International Institute for Software Technology, United Nations University, Macao SAR, China
zbchen@nudt.edu.cn, Z.Liu@iist.unu.edu

Abstract. Compensating CSP (cCSP) is an extension to CSP for modeling long-running transactions. It can be used to specify programs of service orchestration written in a programming language like WS-BPEL. So far, only an operational semantics and a trace semantics are given to cCSP. In this paper, we extend cCSP with more operators and define for it a stable failures semantics in order to reason about non-determinism and deadlock. We give some important algebraic laws for the new operators. These laws can be justified and understood from the stable failures semantics. A case study is given to demonstrate the extended cCSP.

1 Introduction

Long-Running Transactions (LRT) are attracting increasing research attention recently because of their importance in Service-Oriented Computing (SOC) [10]. A transaction in SOC usually lasts for a long period of time, and involves interactions with different organizations. The notion of atomic transaction is too strict for this scenario due to some requirements such as isolation [10]. LRT are therefore introduced to cope with this problem by using compensation to recover from a failure to ensure the required atomicity and consistency.

Industrial service composition languages, such as WS-BPEL [1] and XLANG [14] are now designed and implemented for programming LRT in service orchestration. For specification and verification of LRT, formalisms have been being proposed and they include StAC [4], Sagas [3], cCSP [6], *etc.* Formalisms can provide formal semantics to an industrial language and serve as the foundation for the understanding of LRT and the development of tool support to verification and analysis.

Compensating CSP (cCSP) extends the process calculus of Communicating Sequential Process (CSP) [13] with mechanisms of interruption and recovery from exceptions for describing LRT. The recovery mechanism in cCSP is the same as the backward recovery proposed in Sagas [9]. There are two types of processes in cCSP, and they are called respectively *standard processes* and *compensable processes.* A standard process is a subset of a CSP process extended with exception handling and transaction block. A compensable process specifies the behavior of the recovery when an exception occurs. A trace semantics is presented in [4] and an operational semantics is in [7], and the consistency between them is studied in [12]. However, without non-deterministic (internal) choice and hiding, cCSP is not expressive enough for relating specifications at different levels of abstraction. It is important to note that abstraction (via hiding) is the main source of non-determinism and non-determinism can causes deadlocks when composing processes.

Internal choice and hiding are motivated in the definition of StAC [4,5], a formal notation for LRT that supports synchronized parallel composition, internal and external choices, hiding, and programmable compensation. A serious drawback of StAC compared to cCSP is that StAC does not support compositional reasoning. A thorough comparative study between Sagas [3] and cCSP is presented in [2] and shows two equivalent subsets of them. The paper also compares the policies of the interruption and compensation in Sagas and cCSP, and finds that the revised Sagas [3] is more expressive than cCSP [6]. There is an attempt to extend cCSP [11], but only with synchronized parallel composition.

In this paper, we extend cCSP by bringing back the CSP operators of hiding, internal choice for non-determinism, and synchronized parallel composition for general composition. Accordingly to characterize non-determinism and deadlock, we define a stable failures semantics for the extended language. We show most algebraic laws in the trace semantics of the original cCSP still hold in the stable failures semantics. Also, we show that a few laws that were claimed to hold for the original trace semantics do not hold there, but they hold for the semantics we define in this paper. We study the laws for the newly introduced operators. Due to the page limit, the proofs of the laws are omitted, but they can be found in a technical report [8].

The rest of this paper is organized as follows. Section 2 gives a brief introduction to the syntax and semantics of the original cCSP. Section 3 presents the extended cCSP including the syntax, semantics and laws. Section 4 gives a case study to demonstrate the extended cCSP. Section 5 concludes the paper and reviews the related work.

2 Compensating CSP

The syntax of cCSP [6] is as follows, where P and PP represent a *standard process* and a *compensable process*, respectively.

$$\begin{aligned} P &::= A \mid P\ ;\ P \mid P \Box P \mid P \parallel P \mid \mathit{SKIP} \mid \mathit{THROW} \mid \mathit{YIELD} \mid P \rhd P \mid [PP] \\ PP &::= P \div P \mid PP\ ;\ PP \mid PP \Box PP \mid PP \parallel PP \mid \mathit{SKIPP} \mid \mathit{THROWW} \mid \mathit{YIELDD} \end{aligned}$$

Process A denotes the process that terminates successfully after performing event A. There are three operators on both the standard processes and the compensable processes: sequential composition (;), deterministic choice ($\Box$) and parallel composition ($\parallel$). *SKIP* is the process that immediately terminates successfully. *THROW* indicates the occurrence of an exception, and the process will be interrupted. *YIELD* can terminate successfully or yield to an interrupt from environment to result in an interruption. $P \rhd Q$ executes process Q after an exception is thrown from P, otherwise it behaves like P. $[PP]$ is a transaction block specifying a long-running transaction, in which a compensable process is defined to specify the transaction.

A compensable process is constructed from *compensation pairs* of the form $P \div Q$, where the execution of process Q can compensate the effects after executing P. *SKIPP* immediately terminates successfully without the need to be compensated. *THROWW* throws an exception and *YIELDD* either terminates successfully or yields to an interrupt. We use $\mathcal{P}$ and $\mathcal{PP}$ to denote the set of standard processes and the set of compensable processes, respectively.

2.1 Basic Notations

Let Σ be the set of all the *normal events* that all processes can perform, called *alphabet* of processes, and Σ^* be the set of the finite traces over Σ. In cCSP, three more events $\checkmark$, ! and ? not in Σ are used. Event $\checkmark$, called the *success terminal event*, represents that the process terminates successfully. Event !, called the *exception terminal event*, represents that the trace terminates with an occurrence of an exception. Event ?, called the *yield terminal event*, represents that the execution terminates by yielding to an interrupt from environment. We use $\Omega = \{\checkmark, !, ?\}$ to denote the set of the terminal events, and define $\Sigma^\Omega = \Sigma \cup \Omega$. In addition, we use $s\,\hat{}\,t$ to represent the *concatenation* of traces s and t, and define

- $\Sigma^*_O = \{s\,\hat{}\,\langle\omega\rangle \mid s \in \Sigma^* \wedge \omega \in O\}$: for an $O \subseteq \Omega$.

Let $\Sigma^{*O} = \Sigma^* \cup \Sigma^*_O$, and we call traces in Σ^*_Ω *terminating traces* and traces in $\Sigma^*_{\{\checkmark\}}$ *successfully terminating traces.*

2.2 Trace Semantics

In contract to the CSP convention [13], the trace set of a standard process in cCSP is not prefix closed. The *trace semantic function* $\mathcal{T} : \mathcal{P} \rightarrow \mathbb{P}(\Sigma^*_\Omega)$ assigns

Atomic process For all $A \in \Sigma$, $\mathcal{T}(A) = \{\langle A, \checkmark\rangle\}$

Sequential composition

$$p \;;\; q = \begin{cases} p_1\,\hat{}\,q & p = p_1\,\hat{}\,\langle\checkmark\rangle \\ p & p = p_1\,\hat{}\,\langle\omega\rangle \wedge \omega \neq \checkmark \end{cases} \qquad \mathcal{T}(P \;;\; Q) = \{p \;;\; q \mid p \in \mathcal{T}(P) \wedge q \in \mathcal{T}(Q)\}$$

Choice $\mathcal{T}(P \Box Q) = \mathcal{T}(P) \cup \mathcal{T}(Q)$

Parallel composition

$$p_1\,\hat{}\,\langle\omega_1\rangle \parallel q_1\,\hat{}\,\langle\omega_2\rangle = \{r\,\hat{}\,\langle\omega_1 \& \omega_2\rangle \mid r \in (p_1 \mid\mid\mid q_1)\}$$

$$\mathcal{T}(P \parallel Q) = \{r \mid r \in (p \parallel q) \wedge p \in \mathcal{T}(P) \wedge q \in \mathcal{T}(Q)\}$$

where

ω_1	$\checkmark$	$\checkmark$	$\checkmark$	!	!	?
ω_1	$\checkmark$	?	!	!	?	?
$\omega_1 \& \omega_2$	$\checkmark$	?	!	!	!	?

Exception handling

$$p \rhd q = \begin{cases} p_1\,\hat{}\,q & p = p_1\,\hat{}\,\langle !\rangle \\ p & p = p_1\,\hat{}\,\langle\omega\rangle \wedge \omega \neq ! \end{cases} \qquad \mathcal{T}(P \rhd Q) = \{p \rhd q \mid p \in \mathcal{T}(P) \wedge q \in \mathcal{T}(Q)\}$$

Basic processes $\mathcal{T}(SKIP) = \{\langle\checkmark\rangle\}$, $\mathcal{T}(THROW) = \{\langle !\rangle\}$, $\mathcal{T}(YIELD) = \{\langle ?\rangle, \langle\checkmark\rangle\}$

Fig. 1. The semantics of standard process

each process P a set $\mathcal{T}(P)$ of terminating traces. Fig. 1 shows the definition of $\mathcal{T}$, where p and q are terminating traces, and $p_1 \mid\mid\mid q_1$ represents the set of interleavings of traces p_1 and q_1, whose formal definition can be refereed to Section 2.3 in [13]. The processes in a parallel composition only synchronize on the terminal events, performing other events in an interleaving manner. An exception occurs in the composition if any sub-process throws an exception, and the composition terminates successfully only if both sub-processes do.

A compensable process is defined by a set of pairs of traces, called the *forward trace* and *compensation trace*, respectively. It is necessary to note that the semantics of the sequential composition conforms to the semantics of the classical Sagas [9], and compensation actions are executed in the reverse order of their corresponding forward actions. For example, the forward behavior of $A_1 \div B_1 \; ; \; A_2 \div B_2$ will perform A_1 followed by A_2, but the compensation behavior will perform B_1 after B_2 in case of an exception occurred later. Fig. 2 defines the *trace semantic function* $\mathcal{T}_c : \mathcal{PP} \rightarrow \mathbb{P}(\Sigma_\Omega^* \times \Sigma_\Omega^*)$ of the compensable processes.

To allow a compensable process PP to implicitly yield to an interrupt from the environment at the beginning, in the definition of a compensation pair $P \div Q$ in Fig. 2, the trace pair $(\langle ? \rangle, \langle \checkmark \rangle)$ is included. On the other hand, *YIELD* can be used in any place in a process if one would like to explicitly specify a yield to an interrupt at that place. The semantics of a transaction block $[PP]$ is defined below.

$$\mathcal{T}([PP]) = \{p \,\hat{}\, p' \mid (p \,\hat{}\, \langle ! \rangle, p') \in \mathcal{T}_c(PP)\} \cup \{p \,\hat{}\, \langle \checkmark \rangle \mid (p \,\hat{}\, \langle \checkmark \rangle, p') \in \mathcal{T}_c(PP)\}$$

It says that after an exception occurs, the compensation trace will be executed to recover from the failure. Otherwise, the compensation trace is not executed.

Discussion. In paper [6] some laws are given for the trace semantics. However, our careful investigation finds that some of them do not actually hold there. The

Compensation pair

$$p \div q = \begin{cases} (p, q) & p = p_1 \,\hat{}\, \langle \checkmark \rangle \\ (p, \checkmark) & p = p_1 \,\hat{}\, \langle \omega \rangle \wedge \omega \neq \checkmark \end{cases}$$

$$\mathcal{T}_c(P \div Q) = \{p \div q \mid p \in \mathcal{T}(P) \wedge q \in \mathcal{T}(Q)\} \cup \{(\langle ? \rangle, \langle \checkmark \rangle)\}$$

Compensable sequential composition

$$(p, p') \; ; \; (q, q') = \begin{cases} (p_1 \,\hat{}\, q, q' \; ; \; p') & p = p_1 \,\hat{}\, \langle \checkmark \rangle \\ (p, p') & p = p_1 \,\hat{}\, \langle \omega \rangle \wedge \omega \neq \checkmark \end{cases}$$

$$\mathcal{T}_c(PP \; ; \; QQ) = \{(p, p') \; ; \; (q, q') \mid (p, p') \in \mathcal{T}_c(PP) \wedge (q, q') \in \mathcal{T}_c(QQ)\}$$

Compensable choice $\mathcal{T}_c(PP \Box QQ) = \mathcal{T}_c(PP) \cup \mathcal{T}_c(QQ)$

Compensable parallel composition

$$(p, p') \parallel (q, q') = \{(r, r') \mid r \in (p \parallel q) \wedge r' \in (q \parallel q')\}$$

$$\mathcal{T}_c(PP \parallel QQ) = \{rr \mid rr \in (pp \parallel qq) \wedge pp \in \mathcal{T}_c(PP) \wedge qq \in \mathcal{T}_c(QQ)\}$$

Compensable basic processes

SKIP = *SKIP* ÷ *SKIP*, *THROWW* = *THROW* ÷ *SKIP*, *YIELDD* = *YIELD* ÷ *SKIP*

Fig. 2. The semantics of compensable process

first is $PP; SKIPP = PP$. For example, according to the semantic definition in Fig. 2, the semantics of the process $A \div B$ is $\{(\langle A, \checkmark\rangle, \langle B, \checkmark\rangle), (\langle ?\rangle, \langle \checkmark\rangle)\}$, but the semantics of $A \div B$; $SKIPP$ is $\{(\langle A, \checkmark\rangle, \langle B, \checkmark\rangle), (\langle ?\rangle, \langle \checkmark\rangle), (\langle A, ?\rangle, \langle B, \checkmark\rangle)\}$. The law is not valid because of the extra trace pair $(\langle ?\rangle, \langle \checkmark\rangle)$ added to a compensation pair in the semantic definition.The laws $[P \div Q]{=}P$ and $[P \div Q; THROWW]{=}P; Q$ do not hold either when P does not terminate with the yield terminal event ?. It is because the transaction block will remove the exception terminal event ! of the forward trace. Intuitively, we expect these laws to hold. Indeed, we will see later they become valid in the stable failures semantics in this paper.

3 Extended cCSP and Its Stable Failures Semantics

We extend cCSP with operators of internal and external choices, hiding, renaming and generalized parallel composition for both the standard and compensable processes. The syntax of the extended cCSP is defined as follows, where $A \in \Sigma$, $X \subseteq \Sigma$, and $R \subseteq \Sigma \times \Sigma$.

$$\begin{aligned} P ::= {} & A \mid P \;;\; P \mid P \sqcap P \mid P \Box P \mid P \underset{X}{\parallel} P \mid SKIP \mid THROW \mid YIELD \mid \\ & STOP \mid P \setminus X \mid P[\![R]\!] \mid P \rhd P \mid [PP] \\ PP ::= {} & P \div P \mid PP \;;\; PP \mid PP \sqcap PP \mid PP \Box PP \mid PP \underset{X}{\parallel} PP \mid SKIPP \mid \\ & THROWW \mid YIELDD \mid PP \setminus X \mid PP[\![R]\!] \end{aligned}$$

$P \sqcap Q$ and $P \Box Q$ represent internal and external choices, respectively. In the generalized parallel composition $P \underset{X}{\parallel} Q$, processes P and Q synchronize on the events in X, as well as on the terminal events in Ω. $P \setminus X$ is the process with the events in X being restricted from happening during the execution of P, and $P[\![R]\!]$ the process obtained from P by renaming its events according to the renaming relation R.

We extend the compensable processes similarly by introducing the same operators. The internal and external choices in the compensable processes are made during the execution of the forward behaviors of the sub-processes. PP and QQ in $PP \underset{X}{\parallel} QQ$ synchronize on the events in X between both the forward behaviors and the compensation behaviors of the two sub-processes.

3.1 Semantics of Standard Process

The semantics of a standard process is slightly different from the stable failures semantics of a CSP process in [13], due to the two new terminal events ! and ?. The stable failures model of a standard process P is a pair (T, F), where $T \subseteq \Sigma^{*\Omega}$ is the *trace set* and $F \subseteq \Sigma^{*\Omega} \times \mathbb{P}(\Sigma^{\Omega})$ is the *stable failure set*. The domain of the pairs of traces and failues should satisfy the following axioms.

$$T \text{ is non-empty and prefix closed} \tag{1}$$

$$(s, X) \in F \Rightarrow s \in T \tag{2}$$

$$(s, X) \in F \wedge Y \subseteq X \Rightarrow (s, Y) \in F \tag{3}$$

$$(s, X) \in F \wedge \forall a \in Y \bullet s\,\hat{}\,\langle a \rangle \notin T \Rightarrow (s, X \cup Y) \in F \tag{4}$$

$$s\,\hat{}\,\langle \omega \rangle \in T \Rightarrow (s, \Sigma^{\Omega} \setminus \{\omega\}) \in F,\ \mathit{where}\ \omega \in \Omega \tag{5}$$

$$s\,\hat{}\,\langle \omega \rangle \in T \Rightarrow (s\,\hat{}\,\langle \omega \rangle, X) \in F,\ \mathit{where}\ \omega \in \Omega \wedge X \subseteq \Sigma^{\Omega} \tag{6}$$

In what follows we define the *trace set function* $\mathcal{T_S} : \mathcal{P} \to \mathbb{P}(\Sigma^{*\Omega})$ and the *stable failure set function* $\mathcal{F_S} : \mathcal{P} \to \mathbb{P}(\Sigma^{*\Omega} \times \mathbb{P}(\Sigma^{\Omega}))$ for the standard processes in the extended cCSP.

Atomic and basic processes. Process A can perform event A and terminate successfully. The semantic functions are as follows.

$$\begin{aligned}
\mathcal{T_S}(A) &= \{\langle\rangle, \langle A \rangle, \langle A, \checkmark \rangle\} \\
\mathcal{F_S}(A) &= \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\Omega} \wedge A \notin X\} \cup \{(\langle A \rangle, X) \mid X \subseteq \Sigma^{\Omega} \wedge \checkmark \notin X\} \cup \\
&\quad\ \{(\langle A, \checkmark \rangle, X) \mid X \subseteq \Sigma^{\Omega}\}
\end{aligned}$$

The trace and failure sets of processes *SKIP*, *THROW*, *YIELD* and *STOP* are defined below.

$$\begin{aligned}
&\mathcal{T_S}(\mathit{SKIP}) = \{\langle\rangle, \langle \checkmark \rangle\} \qquad \mathcal{T_S}(\mathit{THROW}) = \{\langle\rangle, \langle ! \rangle\} \\
&\mathcal{T_S}(\mathit{YIELD}) = \{\langle\rangle, \langle \checkmark \rangle, \langle ? \rangle\} \qquad \mathcal{T_S}(\mathit{STOP}) = \{\langle\rangle\} \\
&\mathcal{F_S}(\mathit{SKIP}) = \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\Omega} \wedge \checkmark \notin X\} \cup \{(\langle \checkmark \rangle, X) \mid X \subseteq \Sigma^{\Omega}\} \\
&\mathcal{F_S}(\mathit{THROW}) = \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\Omega} \wedge\, ! \notin X\} \cup \{(\langle ! \rangle, X) \mid X \subseteq \Sigma^{\Omega}\} \\
&\mathcal{F_S}(\mathit{YIELD}) = \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\Omega} \wedge\, ? \notin X\} \cup \{(\langle ? \rangle, X) \mid X \subseteq \Sigma^{\Omega}\} \cup \\
&\qquad\qquad \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\Omega} \wedge \checkmark \notin X\} \cup \{(\langle \checkmark \rangle, X) \mid X \subseteq \Sigma^{\Omega}\} \\
&\mathcal{F_S}(\mathit{STOP}) = \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\Omega}\}
\end{aligned}$$

Internal choice. $P \sqcap Q$ can refuse an event set after performing a trace s if P or Q can refuse the event set after s. The semantic of internal choice is same as that in [13], i.e. the traces and failures of an internal choice are the unions of the traces and failures of its sub-processes, respectively.

$$\mathcal{T_S}(P \sqcap Q) = \mathcal{T_S}(P) \cup \mathcal{T_S}(Q) \qquad \mathcal{F_S}(P \sqcap Q) = \mathcal{F_S}(P) \cup \mathcal{F_S}(Q)$$

It is straightforward to see that $\mathit{YIELD} \sqcap \mathit{SKIP} = \mathit{YIELD}$.

External choice. External choice is different from internal choice on the empty trace ($\langle\rangle$), at which $P \Box Q$ can refuse an event set only if both P and Q refuse it. The failure set of external choice needs to take the terminal events ? and ! into account to make axiom (1) on page 5 hold.

$$\begin{aligned}
\mathcal{T_S}(P \Box Q) &= \mathcal{T_S}(P) \cup \mathcal{T_S}(Q) \\
\mathcal{F_S}(P \Box Q) &= \{(\langle\rangle, X) \mid (\langle\rangle, X) \in \mathcal{F_S}(P) \cap \mathcal{F_S}(Q)\} \cup \\
&\quad\ \{(s, X) \mid (s, X) \in \mathcal{F_S}(P) \cup \mathcal{F_S}(Q) \wedge s \neq \langle\rangle\} \cup \\
&\quad\ \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\Omega} \setminus \{\omega\} \wedge \langle \omega \rangle \in \mathcal{T_S}(P) \cup \mathcal{T_S}(Q) \wedge \omega \in \Omega\}
\end{aligned}$$

The internal and external choices are indistinguishable on the basic processes.

$$\mathit{SKIP} \Box \mathit{YIELD} = \mathit{YIELD} \quad \mathit{SKIP} \Box \mathit{THROW} = \mathit{SKIP} \sqcap \mathit{THROW}$$
$$\mathit{YIELD} \Box \mathit{THROW} = \mathit{YIELD} \sqcap \mathit{THROW}$$

Sequential composition. The definition of sequential composition is different from the classic CSP [13] due to the terminal events ! and ?.

$$\begin{aligned}\mathcal{T}_{\mathcal{S}}(P\ ;\ Q) &= \{s \mid s \in \mathcal{T}_{\mathcal{S}}(P) \cap \Sigma^{*\{!,?\}}\} \cup \{s^\frown t \mid s^\frown\langle\checkmark\rangle \in \mathcal{T}_{\mathcal{S}}(P) \wedge t \in \mathcal{T}_{\mathcal{S}}(Q)\}\\ \mathcal{F}_{\mathcal{S}}(P\ ;\ Q) &= \{(s,X) \mid s \in \Sigma^{*\{!,?\}} \wedge (s, X \cup \{\checkmark\}) \in \mathcal{F}_{\mathcal{S}}(P)\} \cup\\ &\qquad \{(s^\frown t, X) \mid s^\frown\langle\checkmark\rangle \in \mathcal{T}_{\mathcal{S}}(P) \wedge (t,X) \in \mathcal{F}_{\mathcal{S}}(Q)\}\end{aligned}$$

However, the following two laws in [6] still hold here.

$$\mathit{THROW}\ ;\ P = \mathit{THROW} \qquad\qquad \mathit{YIELD}\ ;\ \mathit{YIELD} = \mathit{YIELD}$$

The first law ensures the *exception-stop* semantics that is adopted in many modern languages.

Parallel composition. The parallel composition has to take care of the synchronization of the terminal events. We use $s \underset{X}{\|} t$ to represent the trace set of the synchronization between two traces s and t on X. As well as on the terminal events, s and t need to synchronize on the events in X. The definition of $s \underset{X}{\|} t$ can be referred to that in classical CSP [13] except the synchronization between terminal events, which uses the definition in cCSP (cf. $\omega_1 \& \omega_2$ in Fig. 1).

To define the semantics of $P \underset{X}{\|} Q$, we first define its trace set, and then its failure set. The trace set of $P \underset{X}{\|} Q$ is as follows based on the cases defined above.

$$\mathcal{T}_{\mathcal{S}}(P \underset{X}{\|} Q) = \{u \mid \exists s \in \mathcal{T}_{\mathcal{S}}(P), t \in \mathcal{T}_{\mathcal{S}}(Q) \bullet u \in s \underset{X}{\|} t\} \tag{7}$$

$P \underset{X}{\|} Q$ can refuse an event in $X \cup \Omega$ if either P or Q can. However, because both P and Q can perform the events outside $X \cup \Omega$ independently, $P \underset{X}{\|} Q$ refuses an event outside $X \cup \Omega$ only if both P and Q refuse it. For a failure (s, Y) in P and (t, Z) in Q, the following set is their synchronized failure set under the classical CSP definition.

$$(s, Y) \oplus (t, Z) = \{(u, Y \cup Z) \mid Y \setminus (X \cup \Omega) = Z \setminus (X \cup \Omega) \wedge u \in s \underset{X}{\|} t\} \tag{8}$$

However, this definition has to be modified for the extended cCSP, to take into account the different cases of synchronization on the terminal events in Ω.

– If P or Q cannot perform a terminal event after executing s or t, then $P \underset{X}{\|} Q$ cannot terminate because P and Q need to synchronize on the terminal events. We can use the definition (8) for this case. For example, if Σ is $\{A, B\}$, consider processes A and $B\ ;\ \mathit{THROW}$. We have the failure $(\langle\rangle, \{B, \checkmark, !, ?\})$ of A and the failure $(\langle B\rangle, \{B, \checkmark, ?\})$ of $B\ ;\ \mathit{THROW}$, and thus the failure $(\langle B\rangle, \{B, \checkmark, !, ?\})$ of $A \underset{\{\}}{\|} (B\ ;\ \mathit{THROW})$.

– If both P and Q can terminate, the synchronized terminal event should be removed from the refusal set of the synchronized failure. For example, if Σ is $\{A\}$, consider processes A and $A; \mathit{THROW}$. A has the failure $(\langle A\rangle, \{A, !, ?\})$, and $A\ ;\ \mathit{THROW}$ has the failure $(\langle A\rangle, \{A, \checkmark, ?\})$. We can see that $\checkmark$ is the terminal event A can perform, and $!$ is the terminal event $A\ ;\ \mathit{THROW}$ can perform. The synchronization of these two terminal events is $!$, which should not be contained in the refusal set of the synchronized failure in $A \underset{\{A\}}{\|} (A; \mathit{THROW})$, i.e. $(\langle A\rangle, \{A, \checkmark, ?\})$. If we use the definition (8), the synchronized failure set will contain $(\langle A\rangle, \{A, \checkmark, ?, !\})$, which indicates $A \underset{\{A\}}{\|} (A; \mathit{THROW})$ will deadlock after executing $\langle A\rangle$.

The synchronized failure set of two failures is defined as follows.

$$(s,Y)\oplus(t,Z)=\begin{cases}\{(u,Y\cup Z)\mid Y\setminus(X\cup\Omega)=Z\setminus(X\cup\Omega)\wedge u\in s\underset{X}{\parallel}t\}\\ \qquad\qquad\textbf{if } (s,Y\cup\Omega)\in\mathcal{F}_{\mathcal{S}}(P)\vee(t,Z\cup\Omega)\in\mathcal{F}_{\mathcal{S}}(Q)\\ \\ \{(u,(Y\cup Z)\setminus\Theta)\mid Y\setminus(X\cup\Omega)=Z\setminus(X\cup\Omega)\wedge u\in s\underset{X}{\parallel}t\wedge\\ \qquad\qquad\Theta=rf(\omega_1,\omega_2)\}\\ \qquad\qquad\textbf{otherwise}\end{cases}\tag{9}$$

where ω_1 is the terminal event that P can engage in after performing s, i.e. $\forall(s,Y_1)\in\mathcal{F}_{\mathcal{S}}(P)\bullet Y\subseteq Y_1\Rightarrow(\omega_1\in\Omega\wedge\omega_1\notin Y_1)$, ω_2 is the terminal event that Q can engage in after performing t, and the function rf synchronizing the terminal events is defined as follows, in which $\omega_1\&\omega_2$ is defined in Fig. 1.

$$rf(\omega_1,\omega_2)=\begin{cases}\{\omega_1\&\omega_2\} & \omega_1\in\Omega\wedge\omega_2\in\Omega\\ \{\omega_1\} & \omega_1\in\Omega\wedge\omega_2=\epsilon\\ \{\omega_2\} & \omega_2\in\Omega\wedge\omega_1=\epsilon\\ \{\} & \omega_1=\epsilon\wedge\omega_2=\epsilon\end{cases}\tag{10}$$

If P or Q can terminate with different terminal events after executing a trace, ω_1 or ω_2 may not exist for some failures, e.g. $(\langle\rangle,\{?\})$ in the failure set of the process $SKIP\sqcap THROW$. If ω_1 or ω_2 does not exist, we use ϵ to represent it. Now the failure set of $P\underset{X}{\parallel}Q$ is defined below.

$$\mathcal{F}_{\mathcal{S}}(P\underset{X}{\parallel}Q)=\{(u,E)\mid(u,E)\in(s,Y)\oplus(t,Z)\wedge\\ \exists s,t\bullet(s,Y)\in\mathcal{F}_{\mathcal{S}}(P)\wedge(t,Z)\in\mathcal{F}_{\mathcal{S}}(Q)\}$$

For example, the trace and failure sets of the process $A\underset{\{A\}}{\parallel}(A;THROW)$ in the last example are $\{\langle\rangle,\langle A\rangle,\langle A,!\rangle\}$ and $\{(\langle\rangle,X)\mid X\subseteq\Omega\}\cup\{(\langle A\rangle,X)\mid X\subseteq\{A,\checkmark,?\}\}$ $\cup\{(\langle A,!\rangle,X)\mid X\subseteq\Sigma^{\Omega}\}$, respectively.

The following laws for parallel composition reflect the termination policies in a parallel composition.

$$\begin{array}{ll} YIELD\underset{X}{\parallel}SKIP=YIELD & THROW\underset{X}{\parallel}SKIP\ =THROW\\ THROW\underset{X}{\parallel}YIELD=THROW & THROW\underset{X}{\parallel}THROW=THROW\end{array}$$

If P does not terminate with an yield terminal event, i.e. $\forall s\in\mathcal{T}_{\mathcal{S}}(P)\bullet s\notin\Sigma^*_{\{?\}}$, the parallel composition $\parallel$ without synchronization agrees with the composition $\underset{\{\}}{\parallel}$ and it enjoys the following laws.

$$THROW\parallel P=P\ ;\ THROW$$
$$THROW\parallel(YIELD\ ;\ P)=THROW\sqcap(P\ ;\ THROW)$$

The last law says that a process can be interrupted by an interrupt from the environment, but the interruption does not have priority over other events.

Exception handling. $P\rhd Q$ behaves similarly to $P;Q$, but Q starts to execute only after an exception is thrown in P.

$$\mathcal{T}_{\mathcal{S}}(P \rhd Q) = \{s \mid s \in \mathcal{T}_{\mathcal{S}}(P) \cap \Sigma^{*\{\checkmark,?\}}\} \cup \{s\hat{\ }t \mid s\hat{\ }\langle !\rangle \in \mathcal{T}_{\mathcal{S}}(P) \wedge t \in \mathcal{T}_{\mathcal{S}}(Q)\}$$
$$\mathcal{F}_{\mathcal{S}}(P \rhd Q) = \{(s, X) \mid s \in \Sigma^{*\{\checkmark,?\}} \wedge (s, X \cup \{!\}) \in \mathcal{F}_{\mathcal{S}}(P)\} \cup$$
$$\{(s\hat{\ }t, X) \mid s\hat{\ }\langle !\rangle \in \mathcal{T}_{\mathcal{S}}(P) \wedge (t, X) \in \mathcal{F}_{\mathcal{S}}(Q)\}$$

Laws for exception handling:

$$\begin{aligned} P \rhd \mathit{THROW} &= P & P \rhd (Q \sqcap R) &= (P \rhd Q) \sqcap (P \rhd R)\\ \mathit{THROW} \rhd P &= P & (P \sqcap Q) \rhd R &= (P \rhd R) \sqcap (Q \rhd R)\\ \mathit{SKIP} \rhd P &= \mathit{SKIP} & P \rhd (Q \Box R) &= (P \rhd Q) \Box (P \rhd R)\\ \mathit{YIELD} \rhd P &= \mathit{YIELD} & P \rhd (Q \rhd R) &= (P \rhd Q) \rhd R\\ \mathit{STOP} \rhd P &= \mathit{STOP} \end{aligned}$$

The terminal events do not affect hiding and renaming operators. Thus, their definitions remain the same as those given in the classical CSP.

3.2 Semantics of Compensable Process

The semantics of a compensable process PP is to be defined as a triple (T, F, C), where T and F are the trace and failure sets of the *forward behavior*, and $C \subseteq \Sigma^*_\Omega \times \mathbb{P}(\Sigma^{*\Omega}) \times \mathbb{P}(\Sigma^{*\Omega} \times \mathbb{P}(\Sigma^\Omega))$ defines the *compensation behavior*. The reason for the separation of forward and compensation behaviors is the compensation behavior needs to be recorded during the execution of the forward behavior. An element in C is (s, T^c, F^c), which shows that the behavior defined by the trace set T^c and the failure set F^c can compensate the effects caused by executing the terminating trace s from the forward behavior. Therefore, both the forward behavior (T, F), denoted by PP_f, and the compensation behavior (T^c, F^c) of each element in C, denoted by PP_c, satisfy the axioms of the semantics of the standard processes given in Section 3.1. We can thus overload the semantic functions $\mathcal{T}_{\mathcal{S}}$ and $\mathcal{F}_{\mathcal{S}}$ and the operators on standard processes and apply them to PP_f and PP_c. For examples, $\mathcal{F}_{\mathcal{S}}(PP_f) = F$ and $\mathcal{F}_{\mathcal{S}}(PP_c) = F^c$, and later when we define the semantics of $PP\ ;\ QQ$, we will use the notations $\mathcal{T}_{\mathcal{S}}(PP_f\ ;\ QQ_f)$ and $\mathcal{F}_{\mathcal{S}}(PP_f\ ;\ QQ_f)$ as if PP_f and QQ_f are standard processes. In addition, (T, F, C) is required to satisfy the following axiom.

$$\forall (s, T^c, F^c) \in C \bullet s \in \Sigma^*_\Omega \cap \{s \mid (s, X) \in F\} \qquad (11)$$

It means the trace s of each element in C is a stable terminating trace in the forward behavior.

We define the triple (T, F, C) for a PP by three semantic functions: the *forward trace set* function $\mathcal{T}^c : \mathcal{PP} \rightarrow \mathbb{P}(\Sigma^{*\Omega})$, the *forward failure set* function $\mathcal{F}^c : \mathcal{PP} \rightarrow \mathbb{P}(\Sigma^{*\Omega} \times \mathbb{P}(\Sigma^\Omega))$, and the *compensation behavior set* function $\mathcal{C} : \mathcal{PP} \rightarrow \mathbb{P}(\Sigma^*_\Omega \times \mathbb{P}(\Sigma^{*\Omega}) \times \mathbb{P}(\Sigma^{*\Omega} \times \mathbb{P}(\Sigma^\Omega)))$.

Compensation pair. If the forward behavior terminates successfully, the behavior of Q is recorded such that it can be executed to compensate the effect of P when triggered by an exception later. Otherwise, Q will not be executed. The semantics of a compensation pair $P \div Q$ attaches the successfully terminating trace in the forward behavior, i.e. $s \in T \cap \Sigma^*_{\{\checkmark\}}$, with the trace and failure sets of Q, and the others terminating traces (the traces in $\Sigma^*_{\{!,?\}}$) with those of *SKIP*.

$$\begin{aligned}\mathcal{T}^c(P \div Q) &= \mathcal{T}_{\mathcal{S}}(P) \quad \mathcal{F}^c(P \div Q) = \mathcal{F}_{\mathcal{S}}(P)\\ \mathcal{C}(P \div Q) &= \{(s, T^c, F^c) \mid \exists s \in T \cap \Sigma^*_{\Omega} \bullet\\ &\qquad (s = t\,\hat{}\,\langle\checkmark\rangle \wedge T^c = \mathcal{T}_{\mathcal{S}}(Q) \wedge F^c = \mathcal{F}_{\mathcal{S}}(Q)) \vee\\ &\qquad (s \in \Sigma^*_{\{!,?\}} \wedge T^c = \mathcal{T}_{\mathcal{S}}(SKIP) \wedge F^c = \mathcal{F}_{\mathcal{S}}(SKIP))\}\end{aligned}$$

The compensation behavior set of the compensable processes $STOP \div P$ is empty. In the following, we use $STOPP$ to denote the compensable processes whose forward behaviors are $STOP$. The following two laws hold for compensation pairs.

$$Q_1 = Q_2 \Rightarrow P \div Q_1 = P \div Q_2 \qquad P_1 = P_2 \Rightarrow P_1 \div Q = P_2 \div Q$$

The definitions of the basic compensable processes are the same as those in Section 2.2.

Transaction block. The semantics of a transaction block $[PP]$ can be defined in terms of the semantics of the compensable process PP in the block.

$$\begin{aligned}\mathcal{T}_{\mathcal{S}}([PP]) &= (\mathcal{T}^c(PP) \setminus \Sigma^*_{\{!\}}) \cup\\ &\quad \{s_1 \mid \exists (s, T^c, D^c) \in \mathcal{C}(PP) \bullet s = t\,\hat{}\,\langle !\rangle \wedge s_2 \in T^c \wedge s_1 = t\,\hat{}\,s_2\}\\ \mathcal{F}_{\mathcal{S}}([PP]) &= \{(s, X) \mid s \in \Sigma^* \wedge (s, X \cup \{!\}) \in \mathcal{F}^c(PP)\} \cup\\ &\quad \{(s_1, X_1) \mid \exists (s, T^c, F^c) \in \mathcal{C}(PP) \bullet (s \in \Sigma^*_{\{\checkmark,?\}} \wedge s_1 = s \wedge X_1 \subseteq \Sigma^{\Omega}) \vee\\ &\qquad (s = t\,\hat{}\,\langle !\rangle \wedge (s_2, X_2) \in F^c \wedge s_1 = t\,\hat{}\,s_2 \wedge X_1 = X_2)\}\end{aligned}$$

The compensation behavior of PP will be executed to recover from a failure occurred in the forward behavior. The trace set of $[PP]$ contains the traces of the forward behavior of PP and the traces of compensation behavior. The failure set $\mathcal{F}_{\mathcal{S}}([PP])$ contains the failures of the forward behavior that do not terminate with an exception terminal event. It also includes the failures that extend the exception terminating traces of the forward behavior with the failures of the compensation behavior. Different from the original cCSP, we keep the yield interruption behavior in the semantics of transaction block. The following laws hold.

$$\begin{aligned}[SKIP \div P] &= SKIP & [STOPP] &= STOP\\ [THROW \div P] &= SKIP & [P \div Q] &= P \rhd SKIP\\ [YIELD \div P] &= YIELD & PP_1 = PP_2 &\Rightarrow [PP_1] = [PP_2]\end{aligned}$$

The law $[P \div Q] = P \rhd SKIP$ fixes the problem of the original cCSP pointed out in Section 2.2, i.e. $[P \div Q] = P$ under the assumption that P does not terminate with the yield terminal event.

Sequential composition. In a sequential composition $PP;QQ$, the forward behavior PP_f and the forward behavior QQ_f are composed first, and the compensation behavior PP_c and the compensation behavior QQ_c are composed in the reverse direction, just like the model of Sagas [9].

$$\begin{aligned}\mathcal{T}^c(PP \;;\; QQ) &= \mathcal{T}_{\mathcal{S}}(PP_f \;;\; QQ_f) \quad \mathcal{F}^c(PP \;;\; QQ) = \mathcal{F}_{\mathcal{S}}(PP_f \;;\; QQ_f)\\ \mathcal{C}(PP \;;\; QQ) &= \{(s, T^c, F^c) \mid \exists (s_1, PP_c) \in \mathcal{C}(PP), (s_2, QQ_c) \in \mathcal{C}(QQ) \bullet\\ &\quad (s_1 = t\,\hat{}\,\langle\checkmark\rangle \wedge s = t\,\hat{}\,s_2 \wedge T^c = \mathcal{T}_{\mathcal{S}}(QQ_c; PP_c) \wedge F^c = \mathcal{F}_{\mathcal{S}}(QQ_c; PP_c)) \vee\\ &\quad (s_1 \neq t\,\hat{}\,\langle\checkmark\rangle \wedge s = s_1 \wedge T^c = \mathcal{T}_{\mathcal{S}}(PP_c) \wedge F^c = \mathcal{F}_{\mathcal{S}}(PP_c))\ \}\end{aligned}$$

For example, the compensation behavior of $A_1 \div B_1; A_2 \div B_2$ is $\{(\langle A_1, A_2, \checkmark\rangle, \mathcal{T}_{\mathcal{S}}(B_2; B_1), \mathcal{F}_{\mathcal{S}}(B_2; B_1))\}$, and that of $A_1 \div B_1; \mathit{YIELDD}$ contains two elements: $(\langle A_1, \checkmark\rangle, \mathcal{T}_{\mathcal{S}}(B_1), \mathcal{F}_{\mathcal{S}}(B_1))$ and $(\langle A_1, ?\rangle, \mathcal{T}_{\mathcal{S}}(B_1), \mathcal{F}_{\mathcal{S}}(B_1))$.

Sequential composition satisfies the following laws. In the last two laws, we assume the standard processes P, P_1 and P_2 do not terminate with an exception terminal event.

$$\begin{array}{rcl} \mathit{SKIPP}\ ;\ \mathit{PP} = \mathit{PP} & & \mathit{YIELDD}\ ;\ \mathit{YIELDD} = \mathit{YIELDD} \\ \mathit{PP}\ ;\ \mathit{SKIPP} = \mathit{PP} & & [P \div Q; \mathit{THROWW}] = P; Q \\ \mathit{THROWW}\ ;\ \mathit{PP} = \mathit{THROWW} & & [P_1 \div Q_1; P_2 \div Q_2; \mathit{THROWW}] = P_1; P_2; Q_2; Q_1 \end{array}$$

The second law fixes the right unit problem of the original trace model pointed out in Section 2.2. The fifth law fixes another problem pointed out there and ensures that $[P \div Q; \mathit{THROWW}] = P; Q$ provided that P does not terminate with ?. The last two laws are also valid in the case that the standard processes terminate with ?, and they relax the assumption in the original cCSP that requires all the standard processes terminate successfully.

Internal choice. The semantics of internal choice $\mathit{PP} \sqcap \mathit{QQ}$ is as follows.

$$\mathcal{T}^c(\mathit{PP} \sqcap \mathit{QQ}) = \mathcal{T}_{\mathcal{S}}(\mathit{PP}_f \sqcap \mathit{QQ}_f) \quad \mathcal{F}^c(\mathit{PP} \sqcap \mathit{QQ}) = \mathcal{F}_{\mathcal{S}}(\mathit{PP}_f \sqcap \mathit{QQ}_f)$$
$$\mathcal{C}(\mathit{PP} \sqcap \mathit{QQ}) = \mathcal{C}(\mathit{PP}) \cup \mathcal{C}(\mathit{QQ})$$

For example, the compensation behavior set of $A \div B_1 \sqcap A \div B_2$ is $\{(\langle A, \checkmark\rangle, \mathcal{T}_{\mathcal{S}}(B_1), \mathcal{F}_{\mathcal{S}}(B_1)), (\langle A, \checkmark\rangle, \mathcal{T}_{\mathcal{S}}(B_2), \mathcal{F}_{\mathcal{S}}(B_2))\}$. We have the following laws hold for internal choice.

$$\begin{array}{rcl} \mathit{PP} \sqcap \mathit{PP} = \mathit{PP} & & \mathit{PP} \sqcap (\mathit{QQ} \sqcap \mathit{RR}) = (\mathit{PP} \sqcap \mathit{QQ}) \sqcap \mathit{RR} \\ \mathit{PP} \sqcap \mathit{QQ} = \mathit{QQ} \sqcap \mathit{PP} & & \mathit{PP}\ ;\ (\mathit{QQ} \sqcap \mathit{RR}) = (\mathit{PP}\ ;\ \mathit{QQ}) \sqcap (\mathit{PP}\ ;\ \mathit{RR}) \\ [\mathit{PP} \sqcap \mathit{QQ}] = [\mathit{PP}] \sqcap [\mathit{QQ}] & & (\mathit{QQ} \sqcap \mathit{RR})\ ;\ \mathit{PP} = (\mathit{QQ}\ ;\ \mathit{PP}) \sqcap (\mathit{RR}\ ;\ \mathit{PP}) \end{array}$$

External choice. As in the case of the internal choice, the external choice is made during the forward behavior, but it is the environment to make the choice.

$$\mathcal{T}^c(\mathit{PP} \Box \mathit{QQ}) = \mathcal{T}_{\mathcal{S}}(\mathit{PP}_f \Box \mathit{QQ}_f) \quad \mathcal{F}^c(\mathit{PP} \Box \mathit{QQ}) = \mathcal{F}_{\mathcal{S}}(\mathit{PP}_f \Box \mathit{QQ}_f)$$
$$\mathcal{C}(\mathit{PP} \Box \mathit{QQ}) = \mathcal{C}(\mathit{PP}) \cup \mathcal{C}(\mathit{QQ})$$

For example, $\mathcal{C}(\mathit{STOPP} \Box A \div B)$ equals to $\mathcal{C}(\mathit{STOPP} \sqcap A \div B)$, and they are equal to $\{(\langle A, \checkmark\rangle, \mathcal{T}_{\mathcal{S}}(B), \mathcal{F}_{\mathcal{S}}(B))\}$. But their forward failures sets are different: $\mathcal{F}^c(\mathit{STOPP} \Box A \div B)$ is $\mathcal{F}_{\mathcal{S}}(A)$, and $\mathcal{F}^c(\mathit{STOPP} \sqcap A \div B)$ is $\mathcal{F}_{\mathcal{S}}(A) \cup \mathcal{F}_{\mathcal{S}}(\mathit{STOP})$. The following laws hold for external choice.

$$\begin{array}{rcl} \mathit{PP} \Box \mathit{PP} = \mathit{PP} & & [\mathit{PP} \Box \mathit{QQ}] = [\mathit{PP}] \Box [\mathit{QQ}] \\ \mathit{PP} \Box \mathit{QQ} = \mathit{QQ} \Box \mathit{PP} & & \mathit{PP} \Box (\mathit{QQ} \Box \mathit{RR}) = (\mathit{PP} \Box \mathit{QQ}) \Box \mathit{RR} \\ \mathit{STOPP} \Box \mathit{PP} = \mathit{PP} & & \mathit{PP} \Box (\mathit{QQ} \sqcap \mathit{RR}) = (\mathit{PP} \Box \mathit{QQ}) \sqcap (\mathit{PP} \Box \mathit{RR}) \end{array}$$

Notice that external choice distributes over internal choice. From the laws for internal and external choices, we can see that a transaction block process of a choice between compensable processes equals to a choice between the transaction block processes of the compensable processes.

Parallel composition. In a generalized parallel composition $\mathit{PP} \underset{X}{\parallel} \mathit{QQ}$, the forward behaviors of PP and QQ synchronize on X, so do their compensation behaviors:

$$\mathcal{T}^c(PP \underset{X}{\parallel} QQ) = \mathcal{T}_{\mathcal{S}}(PP_f \underset{X}{\parallel} QQ_f) \quad \mathcal{F}^c(PP \underset{X}{\parallel} QQ) = \mathcal{F}_{\mathcal{S}}(PP_f \underset{X}{\parallel} QQ_f)$$
$$\mathcal{C}(PP \underset{X}{\parallel} QQ) = \{(s, T^c, F^c) \mid \exists (s_1, PP_c) \in \mathcal{C}(PP), (s_2, QQ_c) \in \mathcal{C}(QQ) \bullet$$
$$s \in (s_1 \underset{X}{\parallel} s_2) \wedge T^c = \mathcal{T}_{\mathcal{S}}(PP_c \underset{X}{\parallel} QQ_c) \wedge F^c = \mathcal{F}_{\mathcal{S}}(PP_c \underset{X}{\parallel} QQ_c)\}$$

Here are two examples. First, $[(A \div B_1 \underset{\{A\}}{\parallel} A \div B_2); THROWW] = A; B_1 \parallel B_2$ shows the synchronization between the forward behaviors, and then $A_1 \div B_1 \underset{\{A_1,A_2\}}{\parallel} A_2 \div B_2$ $= STOPP$ demonstrates the case of a deadlock in the forward behavior. The following laws for parallel composition hold.

$$PP \underset{X}{\parallel} QQ = QQ \underset{X}{\parallel} PP$$
$$PP \underset{X}{\parallel} (QQ \underset{X}{\parallel} RR) = (PP \underset{X}{\parallel} QQ) \underset{X}{\parallel} RR$$
$$PP \underset{X}{\parallel} (QQ \sqcap RR) = (PP \underset{X}{\parallel} QQ) \sqcap (PP \underset{X}{\parallel} RR)$$

Furthermore, parallel composition and sequential composition are related by the following laws, where all the standard processes are assumed to terminate successfully.

$$[(P_1 \div Q_1 \underset{X}{\parallel} P_2 \div Q_2)\ ;\ THROWW] = (P_1 \underset{X}{\parallel} P_2); (Q_1 \underset{X}{\parallel} Q_2)$$
$$[(P_1 \div Q_1\ ;\ P_2 \div Q_2) \parallel THROWW] = P_1\ ;\ P_2\ ;\ Q_2\ ;\ Q_1$$
$$[(P_1 \div Q_1\ ;\ YIELDD\ ;\ P_2 \div Q_2) \parallel THROWW] =$$
$$(P_1\ ;\ Q_1) \sqcap (P_1\ ;\ P_2\ ;\ Q_2\ ;\ Q_1)$$
$$[(YIELDD\ ;\ P_1 \div Q_1\ ;\ YIELDD\ ;\ P_2 \div Q_2) \parallel THROWW] =$$
$$SKIP \sqcap (P_1\ ;\ Q_1) \sqcap (P_1\ ;\ P_2\ ;\ Q_2\ ;\ Q_1)$$
$$[P_1 \div Q_1 \parallel P_2 \div Q_2 \parallel THROWW] = (P_1 \parallel P_2)\ ;\ (Q_1 \parallel Q_2)$$
$$[(YIELDD\ ;\ P_1 \div Q_1) \parallel (YIELDD\ ;\ P_2 \div Q_2) \parallel THROWW] =$$
$$SKIP \sqcap (P_1\ ;\ Q_1) \sqcap (P_2\ ;\ Q_2) \sqcap ((P_1 \parallel P_2)\ ;\ (Q_1 \parallel Q_2))$$

The 3rd and 4th laws say that a compensable process in a parallel composition can be interrupted by *YIELDD*, meaning that the process yields to an interrupt from the environment. A compensable process will not be interrupted if no *YIELDD* is used (the 2nd and 5th laws). This is one of the differences from the original cCSP, where a compensable process can implicitly yield to an interrupt from the environment (cf. Section 2.2). We believe that it is more reasonable to let the designer to specify where a compensable process can yield to an interruption from the environment.

Hiding and renaming. Hiding and renaming are defined by the standard hiding and renaming on the forward behavior and the compensation behavior.

$$\mathcal{T}^c(PP \setminus X) = \mathcal{T}_{\mathcal{S}}(PP_f \setminus X) \quad \mathcal{F}^c(PP \setminus X) = \mathcal{F}_{\mathcal{S}}(PP_f \setminus X)$$
$$\mathcal{C}(PP \setminus X) = \{(s, T^c, F^c) \mid \exists (s_1, PP_c) \in \mathcal{C}(PP) \bullet s = s_1 \setminus X \wedge T^c = \mathcal{T}_{\mathcal{S}}(PP_c \setminus X) \wedge$$
$$F^c = \mathcal{F}_{\mathcal{S}}(PP_c \setminus X)\}$$

The renaming semantics is as follows.

$$\mathcal{T}^c(PP[\![R]\!]) = \mathcal{T}_{\mathcal{S}}(PP_f[\![R]\!]) \quad \mathcal{F}^c(PP[\![R]\!]) = \mathcal{F}_{\mathcal{S}}(PP_f[\![R]\!])$$
$$\mathcal{C}(PP[\![R]\!]) = \{(s, T^c, F^c) \mid \exists (s_1, PP_c) \in \mathcal{C}(PP) \bullet s_1\ R\ s \wedge T^c = \mathcal{T}_{\mathcal{S}}(PP_c[\![R]\!]) \wedge$$
$$F^c = \mathcal{F}_{\mathcal{S}}(PP_c[\![R]\!])\}$$

Hiding and renaming satisfy the following laws. In particular, both are distributive among internal choice, but the hiding operator is not distributive among external choice, e.g. $((A;A_1) \div B \Box (A;A_2) \div B) \setminus \{A\}$ is equal to $A_1 \div B \sqcap A_2 \div B$.

$$\begin{array}{rclcrcl}
(PP \setminus X) \setminus Y &=& (PP \setminus Y) \setminus X & & (PP \sqcap QQ)[\![R]\!] &=& PP[\![R]\!] \sqcap QQ[\![R]\!] \\
(PP \setminus X) \setminus Y &=& PP \setminus (X \cup Y) & & (PP \Box QQ)[\![R]\!] &=& PP[\![R]\!] \Box QQ[\![R]\!] \\
(PP \sqcap QQ) \setminus X &=& (PP \setminus X) \sqcap (QQ \setminus X) & & (PP[\![R]\!])[\![R']\!] &=& PP[\![R \circ R']\!] \\
PP \setminus \{\} &=& PP & & & &
\end{array}$$

4 Case Study

This section will give a case study to demonstrate the extended cCSP. It is a business process of an online shop. The system is composed by four parties: a shop, a supplier, a shipper and a bank. The business behavior of each party is compensable, and will be specified by a compensable process.

After receiving a client request (`ReceiveRequest`), the shop contacts its supplier to ask (`SupplierRequest`) whether there exist enough goods. If the storage is not enough (`NotEnough`), the whole process will result in an exception. Otherwise, the shop will make an order (`Order`) of the goods. The shop then contacts the bank for authorizing the credit card of client (`CreditCheck`). If the credit card is valid, the shop processes payment for the client (`Payment`) and informs the supplier (`NotifySupplier`) that the payment is made. For the sake of efficiency, after receiving the notification, the supplier contacts the bank for checking the payment (`PaymentCheck`) and requests the shipper to ship the goods (`ShipRequest`) to client concurrently. If the credit card authorization or payment checking fails (`NotValid, NotPValid`), the whole process will result in an exception. After receiving the shipping request, the shipper schedules a shipping plan (`Schedule`), delivers the goods (`Deliver`) to the client of the shop, and notifies the supplier (`ShipResult`) about the shipping result. The alphabet Σ of the system is given as follows.

```
Σ = {ReceiveRequest, ApologyMail, SupplierRequest, Enough, NotEnough,
     Order, UndoOrder, CreditCheck, Valid, NotValid, Payment, Refund,
     NotifySupplier, PaymentCheck, PValid, NotPValid, ShipRequest,
     NotifyShopShip, ShipResult, Schedule, Deliver, ShipBack}
```

The processes `Shop`, `Supplier`, `Shipper` and `Bank` are specified as follows.

```
Shop = ReceiveRequest ÷ ApologyMail ;
       (SupplierRequest ; (Enough □ (NotEnough ; THROW))) ÷ SKIP ;
       Order ÷ SKIP ; (CreditCheck ; (Valid □ (NotValid ; THROW))) ÷ SKIP ;
       Payment ÷ Refund ; NotifySupplier ÷ SKIP ; NotifyShopShip ÷ SKIP

Supplier = (SupplierRequest ; (Enough ⊓ (NotEnough ; THROW))) ÷ SKIP ;
           Order ÷ UndoOrder ; NotifySupplier ÷ SKIP ;
           ((PaymentCheck ; (PValid □ (NotPValid ; THROW))) ÷ SKIP ||
            (ShipRequest ÷ SKIP ; NotifyShopShip ÷ SKIP)) ; ShipResult ÷ SKIP
```

$$\begin{aligned}\mathtt{Shipper} = {} & \mathtt{ShipRequst} \div \mathtt{SKIP}\ ;\ \mathtt{YIELDD}\ ;\ \mathtt{Schedule} \div \mathtt{SKIP}\ ; \\ & \mathtt{YIELDD}\ ;\ \mathtt{Deliver} \div \mathtt{ShipBack}\ ;\ \mathtt{ShipResult} \div \mathtt{SKIP}\end{aligned}$$

$$\begin{aligned}\mathtt{Bank} = {} & ((\mathtt{CreditCheck}\ ;\ (\mathtt{Valid} \sqcap (\mathtt{NotValid}; \mathtt{THROW}))) \div \mathtt{SKIP}\ ;\ \mathtt{Payment} \div \mathtt{Refund}) \parallel \\ & (\mathtt{PaymentCheck}\ ;\ (\mathtt{PValid} \sqcap (\mathtt{NotPValid}\ ;\ \mathtt{THROW}))) \div \mathtt{SKIP}\end{aligned}$$

The global process (DetailGlobalProcess) is a transaction block of the synchronized parallel composition of the four compensable processes. If the compensable parallel process in the global process results in an exception, a recovery should be taken, e.g. the credit card will be refunded and the shipper will ship back the delivered goods. We can get a more abstract process (AbstractGlobalProcess) by hiding some synchronized events.

$$\mathtt{DetailGlobalProcess} = [\ \mathtt{Shop} \underset{X}{\parallel} \mathtt{Supplier} \underset{X}{\parallel} \mathtt{Shipper} \underset{X}{\parallel} \mathtt{Bank}\ ]$$

$$\begin{aligned}X = \{ & \mathtt{SupplierRequest,\ Enough,\ NotEnough,\ Order,\ CreditCheck,\ Valid,} \\ & \mathtt{NotValid,\ Payment,\ Refund,\ NotifySupplier,\ PaymentCheck,\ PValid,} \\ & \mathtt{NotPValid,\ ShipRequest,\ NotifyShopShip,\ ShipResult}\}\end{aligned}$$

$$\begin{aligned}& \mathtt{AbstractGlobalProcess} = \mathtt{DetailGloablProcess} \setminus X_1 \\ & \quad X_1 = (X \cup \{\mathtt{Schedule}\}) \setminus \{\mathtt{Payment,\ Refund,\ Order}\}\end{aligned}$$

Based on the preceding semantic definitions and laws, the abstract process can be reduced to the following process.

$$\begin{aligned}[\ & \mathtt{ReceiveRequest} \div \mathtt{ApologyMail}\ ;\ \mathtt{SKIPP} \sqcap \mathtt{THROWW}\ ;\ \mathtt{Order} \div \mathtt{UndoOrder}\ ; \\ & \mathtt{SKIPP} \sqcap \mathtt{THROWW}\ ;\ \mathtt{Payment} \div \mathtt{Refund}\ ; \\ & (\mathtt{SKIPP} \sqcap \mathtt{THROWW}) \parallel (\mathtt{YIELDD}\ ;\ \mathtt{Deliver} \div \mathtt{ShipBack})\ ]\end{aligned}$$

It provides an abstract choreography view of the system. According to the semantics, we can get the following results. The proofs of results are omitted due to the space limit and are reported in [8].

- The global process will not deadlock. If we add the event ReceiveRequest to the synchronization set X, a deadlock will happen at the beginning. The reason is: besides Shop, no other process can execute the event ReceiveRequest at the beginning.
- If an exception occurs, ApologyMail is the last event performed in the recovery. From the abstract process, we can see that there are four different cases of recovery: 1) if the storage is not enough, then only ApologyMail will be preformed; 2) if the credit card authorization fails, then the trace $\langle$UndoOrder, ApologyMail$\rangle$ will be performed; 3) if the shipper yields to the exception from the supplier, and the goods delivery will be canceled, then $\langle$Refund, UndoOrder, ApologyMail$\rangle$ will be performed; 4) if the payment checking fails after the goods delivery, then the execution sequence of the recovery is $\langle$ShipBack, Refund, UndoOrder, ApologyMail$\rangle$.

5 Conclusions and Future Work

LRT are important in SOC. It is important that LRT can be formally specified and verified with tool support. The extension of CSP into cCSP is one of the useful attempts in this direction. However, cCSP does not provide the facilities for defining internal choice, hiding and synchronization. This together with the only available trace semantics (and the operational semantics) severely limits the expressive power of the language that it does not specify and reason about non-determinism and dead-lock. In this paper, we have extended cCSP with internal choice, hiding and synchronization in order to be able to specify behavior of LRT at different levels of abstractions. Accordingly, we have provided a stable failures semantics for the extended notation for reasoning about non-determinism and deadlock.

Along with the semantic definitions of the operators, we present the important algebraic laws of the operators. As a by-product of the investigation of the algebraic laws, we have discovered some laws which were claimed to hold but actually do not hold for the trace semantics of cCSP (cf. Section 2.2). In addition, we have proved those laws do hold for our stable failures semantics.

The separation of the forward behavior and compensation behavior in the semantic definition of the compensable processes allows us to understand and analysis the two kinds of computations individually. We can refine and reason about the two parts separately, but the price we are paying is it makes the fixed point theory not clear when recursion is introduced.

From the perspective of theory, future work includes the study of recursion and divergence of LRT, and thus to develop of a full theory of failure-divergence of LRT and their refinements.

Acknowledgments. The work of the first author was supported by the National Basic Research Program of China (973) under Grant No. 2005CB321802 and the NSFC projects under Grant Nos. 60725206, 60970035, 60803042, and the second author acknowledges the support from Macau Science and Technology Development Fund through the project GAVES.

References

1. Alves, A., Arkin, A., Askary, S., Bloch, B., Curbera, F., Goland, Y., Kartha, N., Sterling, König, D., Mehta, V., Thatte, S., van der Rijn, D., Yendluri, P., Yiu, A.: Web services business process execution language version 2.0. OASIS Committee Draft (May 2006)
2. Bruni, R., Butler, M.J., Ferreira, C., Hoare, C.A.R., Melgratti, H.C., Montanari, U.: Comparing two approaches to compensable flow composition. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 383–397. Springer, Heidelberg (2005)
3. Bruni, R., Melgratti, H.C., Montanari, U.: Theoretical foundations for compensations in flow composition languages. In: Palsberg, J., Abadi, M. (eds.) POPL, pp. 209–220. ACM, New York (2005)
4. Butler, M.J., Ferreira, C.: An operational semantics for StAC, a language for modelling long-running business transactions. In: Nicola, R.D., Ferrari, G.L., Meredith, G. (eds.) COORDINATION 2004. LNCS, vol. 2949, pp. 87–104. Springer, Heidelberg (2004)
5. Butler, M.J., Ferreira, C., Ng, M.Y.: Precise modelling of compensating business transactions and its application to BPEL. J. UCS 11(5), 712–743 (2005)

6. Butler, M.J., Hoare, C.A.R., Ferreira, C.: A trace semantics for long-running transactions. In: Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.) Communicating Sequential Processes. LNCS, vol. 3525, pp. 133–150. Springer, Heidelberg (2005)
7. Butler, M.J., Ripon, S.: Executable semantics for compensating CSP. In: Bravetti, M., Kloul, L., Zavattaro, G. (eds.) EPEW/WS-EM 2005. LNCS, vol. 3670, pp. 243–256. Springer, Heidelberg (2005)
8. Chen, Z., Liu, Z.: An extended cCSP with stable failures semantics. Technical Report 431, UNU-IIST (2010)
9. Garcia-Molina, H., Salem, K.: SAGAS. In: Dayal, U., Traiger, I.L. (eds.) SIGMOD Conference, pp. 249–259. ACM Press, New York (1987)
10. Little, M.C.: Transactions and web services. Commun. ACM 46(10), 49–54 (2003)
11. Ripon, S.: Extending and Relating Semantic Models of Compensating CSP. PhD thesis, University of Southampton (2008)
12. Ripon, S., Butler, M.J.: PVS embedding of cCSP semantic models and their relationship. Electr. Notes Theor. Comput. Sci. 250(2), 103–118 (2009)
13. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall PTR, Upper Saddle River (1997)
14. Thatte, S.: XLANG web services for business process design (2001)

Preference and Non-deterministic Choice

Bill Stoddart[1], Frank Zeyda[2], and Steve Dunne[1]

[1] University of Teesside (UK)
[2] University of York (UK)

Abstract. After reviewing a previously reported relational model of reversible computing, in which non-deterministic choice may represent provisional choice subject to revision by backtracking, we narrate our attempts to express preference within choice. We begin our investigations by recalling Nelson's extensions to Dijkstra's calculus, and considering his biased-choice operator as a model of preference. However, we find that this is too short-sighted for our purpose, and we outline the necessity of incorporating a notion of continuation. After considering how this might be achieved in a predicate-transformer approach, we adopt instead a prospective-value semantics that is easily extensible to probabilistic choice; here, we find a clue that helps us to obtain a first formulation of preference. Our formulation, however, takes us into a world of non-monotonic computations, and we are motivated to move on. We look for further inspiration within the execution structures of our reversible virtual machine which provides a construct that records all results of a backtracking search. We add a modified version that records results sequentially, and take its description as the basis for a "temporal order of continuations" semantics, to which we add implementor's choice, which is now quite distinct from provisional choice. We give a refinement relation and prove the monotonicity of the new semantics and its consistency with respect to our previous prospective-values formalism.

Keywords: reversible computing, semantics, backtracking, continuations.

1 Introduction

The theory of reversible computation seeks to organise computations so as to minimise their necessary energy requirements. The link between power consumption and computation comes from the link between energy and information. This tells us, via the theory of Landauer Erasure, that the destruction of information during computation presents a need to consume energy: energy conserving systems are physically reversible, and do not permit a single present state to be arrived at from multiple past states [6]. Reversible computation seeks to minimise information erasure by avoiding the compression of multiple past states into a single state, such as normally happens on execution of an assignment statement. Indeed, such erasure cannot be totally avoided, but, as the analysis of Bennett has shown [1,3] it can be limited to an initialisation phase. For the architecture we envisage, this initialisation would include the writing of a memory area used as a history stack, h, with zero values. This area is subsequently used to retain information that would normally be erased. Using this technique we can perform any assignment $x := e$ without erasure (i.e. using only operations which have right inverses) as follows.

assignment	$h(i-1)$	$h(i)$	x
	?	0	x_0
$h(i) := h(i) + e$	?	e	x_0
$x, h(i) := h(i), x$	?	x_0	e
$i := i + 1$	x_0	0	e

The above, as well as demonstrating the possibility of an assignment statement free from data erasure, also affords us the luxury of programming reversible computations with a normal range of assignment statements rather than just those that are intrinsically reversible (e.g. $x := x + e$ where x is not free in e). Indeed we can perform such transformations for all sequential language structures, and retain all our familiar sequential programming constructs [14].

Nevertheless, in order to reuse the history stack locations thus consumed we need to sometimes reverse our computations, and to this end we introduce a new program structure $S \diamond E$ which runs program S, evaluates expression E, then reverses execution. Thus, for deterministic S, $S \diamond E$ performs an evaluation of E after S without incurring any of the state-changing effects of S. Details of how this is arranged on an implementation platform have been reported in [11,12], which are papers describing the Reversible Virtual Machine (RVM) we use as an implementation platform for our experiments in reversible language design. On our execution platform, we interpret non-deterministic choice as a provisional choice, subject to revision if it leads to an infeasible continuation. We interpret the stand-alone guarded command $g \to S$ as a command that will execute S if g is true, but will reverse if g is false. We thus obtain a sequential programming language with backtracking based on choice and guard. Where S is non-deterministic, it may lead to a number of different prospective values for E, and, using Hehner's Bunch Theory, we interpret $\{S \diamond E\}$ as the set of "prospective values" that E could take after S.

Our approach to reversible computation exploits reversibility to provide new execution structures, and simpler memory management (e.g. garbage collection on reverse execution). In this way we hope to offset the added complexity required by reversible computation, illustrated by the analysis of the reversible assignment given above, and necessarily present at all levels — the logic gates of a reversible computer, for example, must equally function without information erasure, requiring designs such as the Toffoli gate [3].

We can also look to reversibility for some new approaches to program semantics. For example, the rules for $S \diamond E$ give us the same relational semantics of conjunctive computations as the wp calculus with the law of the excluded miracle being revoked. This semantics is not rich enough, however, to express all properties of the underlying computations it models, and these properties are crucially exploited by our execution platform, the RVM. One such property is preference, and others will emerge in the course of our discussions.

The rest of the paper is organised as follows. Section 2 presents mathematical preliminaries. Section 3 considers the biased choice introduced in Nelson's generalisation of Dijkstra's calculus. Section 4 constructs a definition of preferential choice which borrows techniques from the definition of probabilistic choice. Section 5 considers the lack of conjunctivity and monotonicity properties of the new choice operator. Section 6 defines a semantics based on the temporal order of continuations, which provides a distinction between implementor's choice and the provisional choice used in backtracking. We prove monotonicity for the new semantics and establish its consistency with the prospective-value approach.

2 Preliminaries

We employ Hehner's bunch theory to describe our prospective-value (pv) semantics. Hehner points out [5] that set theory, which is foundational in mathematics, gives us, simultaneously, collection and packaging. For modelling purposes it is interesting to treat these independently. To this end we think of the content of a set as having a mathematical rather than a syntactic existence, and we call it a bunch. The content of the set $\{1, 2\}$ is the bunch $1, 2$. The comma in this expression is now an operator, known as bunch union, rather than merely syntax. The content of a set A is written as $\sim A$ where $\sim$ is the unpacking operator. The content of the empty set is known as **null**. More exactly, we follow a multi-sorted approach with a different empty set, and different **null** for each type, and we give such types by subscripting where they cannot be deduced contextually.

The lifting of bunch comma to the status of an operator allows it an ubiquity which, given the other uses of commas in our notations, requires some notational adjustments. We no longer use (a, b) for an ordered pair, using $a \mapsto b$ instead. We retain $f(a, b)$ as a notation for the application of a function to an argument pair, using $g((a, b))$ when we want to apply a function of one argument to a bunch. In this latter case the application is "bunch lifted": $g((a, b)) = g(a), g(b)$. Similarly, we have bunch lifting of infix operators, as in $(1, 2)+(3, 4) = 1+3, 1+4, 2+3, 2+4$. We write $A : B$ to assert that the elements of A are all included in B.

We define the guarded bunch $g \multimap E$ as equal to E when its guard g holds and **null** otherwise. We use an improper bunch $\bot$ (more exactly an improper bunch $\bot_T$ for each type) to represent non-termination in a total-correctness interpretation. We define the preconditioned bunch $P \mid E$ to be equal to E when its precondition P holds, and equal to $\bot$ otherwise. For any bunch A we have $\textbf{null} : A$ and $A : \bot$. Bunches of a given type form a complete lattice under reverse bunch-inclusion, with **null** as its top and $\bot$ as its bottom element.

The expression $\oint x \bullet E$ represents the bunch of all values E can take as x ranges over its type, which is assumed to be inferred from E. For example, $\oint x \bullet (0 \leqslant x \wedge x < 5) \multimap 2x$ is the bunch of even natural numbers less than ten. Bound variables in our theory range over elements, that is singleton bunches.

A model for bunch theory has been formulated by Morris and Bunkenburg [7]. For constant terms, each value V in our universe of bunches has, as its denotation, the set whose elements are the elements of V. Thus $1, 2$ has denotation $\{1, 2\}$ and so on.

Using bunch theory, we can express the following rules for $S \diamond E$ as S ranges over the basic syntactic constructs of our language.

$\mathbf{skip} \diamond E$	$=$	E	skip
$x := F \diamond E$	$=$	$E[F/x]$	assignment
$P \mid S \diamond E$	$=$	$P \mathbin{\rule{2pt}{1em}} (S \diamond E)$	precondition
$g \to S \diamond E$	$=$	$g \rightarrowtriangle (S \diamond E)$	guard
$S \;\square\; T \diamond E$	$=$	$(S \diamond E), (T \diamond E)$	choice
$S \,;\, T \diamond E$	$=$	$S \diamond T \diamond E$	sequential composition
$@x \bullet S \diamond E$	$=$	$\oint x \bullet S \diamond E$	unbounded choice

Assuming S operates on a state variable, or variable list, s we can obtain the predicate $\mathbf{prd}(S)$ that relates initial and final values of s (with final values being dashed) as $\mathbf{prd}(S) = s' : S \diamond s$. The set of before states from which S is not guaranteed to terminate is $\mathbf{nonpre}(S) = \{s \mid S \diamond s = \bot\}$, and the relation which maps before states of S to after states is $\mathbf{rel}(S) = \{s, s' \mid \mathbf{prd}(S)\}$.

We thus have the semantics of a relational model. For terminating computations, sequential composition of operations corresponds to composition of their respective relations: $\mathbf{rel}(S_1 \,;\, S_2) = \mathbf{rel}(S_1) \fatsemi \mathbf{rel}(S_2)$. The effect of a backtracking search is modelled by relational composition discarding partial paths.

This model, however, does not capture all we can usefully know about choice. The RVM provides a provisional choice structure in which the first choice is always tried first. This enables us to order choices according to search heuristics, but the idea of preference inherent in our implementation platform contains more information than can be grafted onto the relational model. Thus the use of search heuristics, and a deterministic description of cut, have been beyond the scope of our formal analysis.

We finish this section with a remark on other notations and precedence. We write $[P]$ to assert that a predicate P is universally true. Our precedence rules give highest priority to expression connectives, followed by logical connectives and then program connectives; in descending order precedence is

$$(*\ /)\ \ (+\ -)\ \ {}_p{+}\ \ \sim\ \ \times\ \ \frown\ \ \cup\ \ \cap\ \ \setminus\ \ \mapsto\ \ \rightarrowtriangle\ \ \mathbin{\rule{2pt}{1em}}\ \ ,\ \ (<\ \leqslant\ >\ \geqslant)\ \ (:\ =\ \neq\ \in\ \notin)$$

$$\neg\ \ \wedge\ \ \vee\ \ \Rightarrow\ \ \Leftrightarrow\ \ :=\ \ \rhd\ \ \square\ \ \sqcup\ \ {}_p{\oplus}\ \ \sqcap\ \ \to\ \ ;\ \ .\ \ \bullet\ \ \diamond\ \ \nabla\ \ (\widehat{=}\ \ =\ \ :\ \ \equiv)$$

The large equals and bunch containment symbols have the same meaning as, but lower precedence than, their smaller equivalents.

3 Nelson's Biased Choice

Non-determinism was originally proposed by Floyd [4] to model backtracking search, but has become even more important as an abstraction tool, allowing us to describe *what* a program should do without providing details of *how* it should do it. As an example of a problem in which both uses are significant, consider the Knight's Tour [13]. The specification says the program will return a path that forms a Knight's tour, but does not say which path. This is non-determinism used as an abstraction tool. Within the implementation, non-determinism represents provisional choice, subject to revision if it leads to infeasibility.

A classic paper that pays some attention to the use of non-deterministic choice to provide backtracking is *An Extension of Dijkstra's Calculus* [8] which is a general correctness treatment of the wp calculus in which Dijkstra's "Law of The Excluded Miracle" is dropped, allowing possibly infeasible programming statements to appear.

Nelson defines a biased choice, which in our notation is expressed as

$$S \boxplus T \;\widehat{=}\; S \;\square\; \neg\mathbf{fis}(S) \rightarrow T$$

This chooses its first operand unless that is infeasible, in which case it chooses the second operand. The feasibility of an operation, which we require for this definition, is defined as the inability of the operation to achieve the impossible:

$$\mathbf{fis}(S) \;\widehat{=}\; \neg\mathrm{wp}(S, \mathit{false})$$

Nelson considers biased choice as a fundamental program connective, and uses it to define a loop structure:

$$\mathbf{do}\; S \;\mathbf{od} \;\widehat{=}\; \mu\, X \bullet (S\,;X) \boxplus \;\mathbf{skip}$$

and considerations of the monotonicity properties of biased choice reveal that it is the Egli-Milner order, rather than refinement ordering that must be used in the corresponding fixed-point treatment.

Looking for an application of this choice in the paper, we find the following: "As an example of the power of clairvoyant non-determinism, we define a command E that parses simple arithmetic expressions, assuming we are given a procedure *Id* that parses identifiers, and procedures *Oplus* and *Otimes* which parse the tokens for the operators $+$ and $\times$. By parsing a syntactic category we mean to accept from the input the longest legal instance of the category, or failing if no prefix of the input belongs to the category. E is defined recursively:"

$$E \;\widehat{=}\; (E\,; \mathit{Oplus}\,; E) \boxplus (E\,; \mathit{Otimes}\,; E) \boxplus \mathit{Id}$$

Preference is used to make the precedence of addition lower than that of multiplication. However, our intuition is that this example is not a description of code, since execution would choose the left recursion at every stage. Indeed, recalling that general correctness allows the description of a definitively non-terminating program **loop**, which has the properties $\mathbf{loop}\,; S = \mathbf{loop}$ and $\mathbf{loop} \boxplus S = \mathbf{loop}$, we see that we can obtain **loop** as a solution to the defining equation for E. Indeed, as **loop** is the bottom element of the Egli-Milner order, it is *the* solution. For our total-correctness calculus, a similar argument would show the solution of the equation to be **abort**. Nelson comments that "an implementation of clairvoyant non-determinism is required to choose an execution that "succeeds" for some notion of success". Here, that notion must imply the avoidance of choices leading to non-termination. Clairvoyance in our formalism, however, as implemented through reversible computation, is based only on the avoidance of infeasibility: "the demon abhors a miracle". Using this paradigm, backtracking parsers whose structure directly mirrors an underlying grammar in the spirit of Nelson's example can be written for grammars which are not left-recursive, and they can express the necessary preferences using biased choice.

As far as the more general use of biased choice to express preference is concerned, we see that it has no possibility of reacting to infeasibility that becomes apparent (in operational terms) after the left component of the biased choice has terminated. We illustrate this in the following example.

$$S \ \widehat{=}\ x := 1 \boxplus x := 2 \quad \text{and} \quad T \ \widehat{=}\ x = 2 \rightarrow \mathbf{skip}$$

We would like S, executed by itself, to be $x := 1$ but when placed in a context where continued execution based on the choice of S will lead to infeasibility, we require the choice to be revised; thus we require that $S\,;T$ is $x := 2$.

We certainly have S equivalent to $x := 1$ but a simple calculation in our pv calculus reveals that $S\,;T \diamond E = \mathbf{null}$, i.e. that $S\,;T$ has no after states and is thus infeasible.

A final consideration for deciding whether to include biased choice in the repertoire of our RVM is its ease of implementation. We have seen that biased choice does not enable the continuation of the program to exercise any revising control, and we therefore wonder how this particular effect could be implemented. In executing a program of the form $S \boxplus T\,; U$, any infeasibility in S causes backtracking which results in a revised choice of T. However, once S has terminated and U is executing, this behaviour must change, and if execution reverses from within U then a revision of our biased choice must *not* be made.

To achieve this effect we could probe the feasibility of S by evaluating $\{S \diamond_1 x\}$. This yields an empty set if S is infeasible, and otherwise yields a unit set containing the value of x found after the first run through S. A candidate for an *executable* definition of $S \boxplus T$ is

$$S \boxplus T \ \widehat{=}\ \mathbf{if}\ \{S \diamond_1 x\} \neq \emptyset\ \mathbf{then}\ S\ \mathbf{else}\ T$$

Here S is executed in the condition clause to probe its feasibility, then the effect of S is undone by reverse execution, before possibly executing S again. A more efficient implementation might be possible, but at the expense of complicating the internal structure of the RVM. So although biased choice keeps us within the relational model, its implementation imposes some added complexity compared to the provisional non-deterministic choice already implemented on the RVM, and, in addition, its ability to express preference is limited to its use at the top level of a sequential program. These considerations, among others, motivate us to reject it as a candidate for preferential choice, and to continue our search.

4 Preference and Probabilistic Choice

Since the wp calculus has proved an effective tool for supporting program development methodologies, we first look here to see what would be involved in expressing the concept of preference we require. Suppose that, whilst performing an analysis of a term $\mathrm{wp}(S, Q)$, we find that we need to analyse $\mathrm{wp}(T, R)$, where T is some component of S, and that after T has terminated some continuation C will complete the execution of S. Thus $R = \mathrm{wp}(C, Q)$. In deciding whether to prefer T to some other choice, we need a formulation that will allow us to choose T *unless it leads to an infeasible computation.* Note that it is not just the feasibility of T itself that concerns us, but rather the feasibility of T together with its continuation C. This combination is feasible if it cannot establish *false*, i.e. if $\neg\,\mathrm{wp}(T\ ;\ C, \mathit{false})$. We need different wp analyses to establish whether the required postcondition is met and whether it is met in a non-vacuous (non-magical) way. We would also need to explicitly extract the continuation C. Since this approach appears clumsy, we look at pv semantics as an alternative, as this will provide us with the means to distinguish substantial and vacuous achievements of a postcondition. For example if we numerotize our predicates with the notation $|P| \mathrel{\widehat=} \textbf{if}\ \ P\ \ \textbf{then}\ \ 1\ \ \textbf{else}\ \ 0$ then we have $S \diamond |P| = \textbf{null}$ where S is infeasible, and where S is feasible we have $S \diamond |P| = 1$ where S will establish P, $S \diamond |P| = 0$ where S will establish $\neg\, P$, $S \diamond |P| = (0, 1)$ where S will establish either P or $\neg\, P$, and $S \diamond |P| = \bot$ where S is not certain to terminate.

Our pv semantics extends smoothly to a probabilistic calculus. Furthermore, probabilistic choice on the RVM (and in our formalisms) is subject to revision by backtracking, and thus presents itself as a candidate for expressing preference. Our expectation calculus for probabilistic choice (including its combination with non-deterministic choice) is given in [9]; here we resume the details required for the current investigation.

We add to our language a probabilistic choice $S\ {}_p\!\oplus\ T$ which makes a provisional choice of S with probability p and of T with probability $1-p$. If the choice leads to infeasibility, it is revised. This is is a rather operational description, but serves to emphasise that not only the feasibility of S or T matters for revision of a choice to occur, but the feasibility of S or T followed by their continuations.

With the addition of probabilism, $S \diamond E$, considered as a term in an executable language, may take different values on different runs. This is suggestive of the intuitive idea of a random variable (though formally a random variable is a function). We use the non-compositional notation $\varepsilon(S \diamond E)$ to express the expected value of $S \diamond E$. The expectation arising from a probabilistic choice where both choices and their continuations are feasible is

$$\varepsilon(S\ {}_p\!\oplus\ T) = p * \varepsilon(S \diamond T) + (1-p) * \varepsilon(T \diamond E)$$

but that will not give us the properties associated with revision of choice due to backtracking. To formulate this we call on bunch notation, and first formulate a weighted addition operator which adjusts to null arguments (we recall that normally, **null** acts as an annihilator, with $x + \textbf{null} = \textbf{null}$). We define:

$$E\ {}_p\!+ F\ \mathrel{\widehat=}\ p * E + (1-p) * F,\ E = \textbf{null} \rightarrowtriangle F,\ F = \textbf{null} \rightarrowtriangle E$$

The RHS of this definition is a bunch consisting of three syntactic items. If neither of E, F is empty the first term gives the value of the bunch; otherwise the first term has a null value, and the value of the expression is determined by the second or third term, at most one of which can be non-null.

We can then give the defining equation for the expectation of a probabilistic choice under the assumption $0 < p < 1$:

$$\varepsilon(S \;{}_p\!\oplus\; T \diamond E) = \varepsilon(S \diamond E) \;{}_p\!+ \varepsilon(T \diamond E)$$

We will need two further properties of the expectation calculus, the first deals with sequential composition:

$$\varepsilon(S\,;T \diamond E) = \varepsilon(S \diamond \varepsilon(T \diamond E))$$

and the second states that for any program S not involving probabilistic choice we have $\varepsilon(S \diamond E) = S \diamond E$.

Now let us see what happens to our little example when the choice is probabilistic; we have to evaluate

$\varepsilon(x := 1 \;{}_p\!\oplus\; x := 2\,; x = 2 \rightarrow \textbf{skip} \diamond x) \;=\;$ "seq comp rule"
$\varepsilon(x := 1 \;{}_p\!\oplus\; x := 2 \diamond \varepsilon(x = 2 \rightarrow \textbf{skip} \diamond x)) \;=\;$ "by absence of probabilistic choice in the second argument and application of pv guard and skip rules"
$\varepsilon(x := 1 \;{}_p\!\oplus\; x := 2 \diamond x = 2 \multimap x) \;=\;$ "prob choice"
$\varepsilon(x := 1 \diamond x = 2 \multimap x) \;{}_p\!+ \varepsilon(x := 2 \diamond x = 2 \multimap x) \;=\;$ "by absence of prob choice and application of pv assignment and guard rules"
$\textbf{null} \;{}_p\!+ 2 \;=\;$ "by weighted addition with null argument" 2

Now, by setting p close to 1 we can use the probabilistic choice $S \;{}_p\!\oplus\; T$ to show a preference for S, and, as we require, this preference will be revised if the preferred choice is infeasible. However, we cannot just set p to 1 and obtain a preferential-choice operator, since in this case probabilistic choice collapses and we are left with just S [9].

However, it seems we could use the same kind of guarded-bunch formalism in expressing preference as in expressing probabilistic choice, namely by introducing a preferential choice $[>$, which prefers its first operand, with the property

$$S \;[>\; T \diamond E \;=\; S \diamond E, (S \diamond E = \textbf{null}) \multimap T \diamond E$$

We further see a possibility to simplify this definition by formulating the concept of preference within bunch notation by first defining a bunch preference operator $\gg$ by virtue of $E \gg F \mathrel{\widehat{=}} E, (E = \textbf{null}) \multimap F$, so that the defining rule for preferential choice becomes $S \;[>\; T \diamond E = (S \diamond E) \gg (T \diamond E)$.

5 Preference and Monotonicity

Standard calculi for sequential programs, such as Dijkstra's GCL, Abrial's GSL, Hoare-He Designs, and Dunne's Prescriptions, describe *conjunctive* computations. In wp terms this means that $\mathrm{wp}(S, Q \wedge R) \Leftrightarrow \mathrm{wp}(S, Q) \wedge \mathrm{wp}(S, R)$. This property is a specialisation of a more general property, that of monotonicity, which is defined as $[Q \Rightarrow R] \Rightarrow (\mathrm{wp}(S, Q) \Rightarrow \mathrm{wp}(S, R))$. Angelic computations are *disjunctive* rather than conjunctive, and disjunctivity, like conjunctivity, implies monotonicity. Where both angelic and demonic computations are accommodated in a calculus, (requiring multi-relations for the associated model) the resulting computations are still monotonic [2].

It may therefore be surprising, to some readers, to find that not all formal descriptions of code are conjunctive or even monotonic, but such, indeed, is what we claim to be the case for computations involving preference as formulated in the previous section.

To present these ideas we formulate the pv equivalent of monotonicity as $[E : F] \Rightarrow (S \diamond E) : (S \diamond F)$. We prove monotonicity for standard pv semantics, (i.e. excluding preferential choice). We appeal to the following lemmas.

Lemma 1. $x : S \diamond E \;\equiv\; \neg\,\mathrm{wp}(S, \neg\, x : E)$

A proof is given in [10].

Lemma 2. $[Q \Rightarrow R] \Rightarrow (\neg\,\mathrm{wp}(S, \neg\, Q) \Rightarrow \neg\,\mathrm{wp}(S, \neg\, R))$

Proof. We assume $[Q \Rightarrow R]$ and thus by predicate logic $[\neg\, R \Rightarrow \neg\, Q]$.
Then by wp monotonicity $\mathrm{wp}(S, \neg\, R) \Rightarrow \mathrm{wp}(S, \neg\, Q)$
and again by predicate logic $\neg\,\mathrm{wp}(S, \neg\, Q) \Rightarrow \neg\,\mathrm{wp}(S, \neg\, R)$.

Theorem 1. ***Monotonicity of pv semantics***

$$[E : F] \Rightarrow (S \diamond E) : (S \diamond F)$$

Proof. We assume $[E : F]$ and thus for arbitrary x' that $x' : E \Rightarrow x' : F$. We then show $x' : (S \diamond E) \Rightarrow x' : (S \diamond F)$ as below.

$x' : (S \diamond E) \;=\;$ "by Lemma 1"
$\neg\,\mathrm{wp}(S, \neg\, x' : E) \;\Rightarrow\;$ "by Lemma 2"
$\neg\,\mathrm{wp}(S, \neg\, x' : F) \;=\;$ "by Lemma 1"
$x' : (S \diamond F)$ □

However, the counterexample below shows that pv monotonicity does not hold when pv semantics is extended with the preferential choice operator [>.

$\mathbf{null} : x$ so by pv monotonicity we expect $(S \diamond \mathbf{null}) : (S \diamond x)$
but for $S \mathrel{\widehat{=}} \mathbf{skip} \;[>\, \mathbf{abort}$ we have
$S \diamond \mathbf{null} = (\mathbf{skip} \diamond \mathbf{null}) \gg (\mathbf{abort} \diamond \mathbf{null}) \;=\;$ "evaluating terms"
$\mathbf{null} \gg \bot \;=\;$ "by definition of bunch preference" $\bot$

whereas

$S \diamond x = (\mathbf{skip} \diamond x) \gg (\mathbf{abort} \diamond x) \;=\;$
$x \gg \bot \;=\;$ "by definition of bunch preference" x

To see the scope of the problem posed by losing pv monotonicity, let us recall how a fixed-point semantics for a language defined in terms of pv semantics is established. We need to confirm that recursive programs are soundly based, and we can do this by appealing to fixed-point theory. We need to establish a partial order between programs, and one that will serve our purpose is $S \sqsubseteq_{\text{pv}} T \mathrel{\widehat{=}} (T \diamond E) : (S \diamond E)$ for arbitrary E. Indeed, under this order our programs (excluding the [> operator) form a complete lattice. Any recursive program P then has an associated monotonic functional F such that $P = F(P)$. We call this fixed-point (fp) monotonicity since it is the property needed for us to appeal to the appropriate Knaster-Tarski least fixed-point theorem. To demonstrate fp monotonicity, we appeal to monotonicity of the program connectives; we have to show, e.g. that if $S \sqsubseteq_{\text{pv}} S'$ and $T \sqsubseteq_{\text{pv}} T'$ then $g \rightarrow S \sqsubseteq_{\text{pv}} g \rightarrow S'$, $S \ \Box \ T \sqsubseteq_{\text{pv}} S' \ \Box \ T'$, $S \ ; \ T \sqsubseteq_{\text{pv}} S' \ ; \ T'$, and so on. By way of example, in establishing $S \ ; \ T \sqsubseteq_{\text{pv}} S' \ ; \ T'$ we appeal to pv monotonicity as follows.

By the pv rule for sequential composition we have $S' ; T' \diamond E = S' \diamond T' \diamond E$. Now since $(T' \diamond E) : (T \diamond E)$ from the assumption $T \sqsubseteq_{\text{pv}} T'$ then by pv monotonicity $(S' \diamond T' \diamond E) : (S' \diamond T \diamond E)$. And again, from the assumption $S \sqsubseteq_{\text{pv}} S'$ and pv monotonicity $(S' \diamond T \diamond E) : (S \diamond T \diamond E)$. Thus by transitivity of bunch inclusion and the rule for pv sequential composition $(S' \ ; \ T' \diamond E) : (S \ ; \ T \diamond E)$, which by the definition of pv refinement gives $S \ ; \ T \sqsubseteq_{\text{pv}} S' \ ; \ T'$.

The loss of pv monotonicity has serious consequences for the construction of a fixed-point semantics. Also, having gone beyond the normal relational model of total correctness, it is not immediately clear what model to now adopt, or what a suitable ordering relationship would be for refinement or fixed-point semantics. These considerations prompt us to consider an alternative approach. A further motivation is the possibility of obtaining a more complete description of the preference already present in the principal implementation of choice within the Reversible Virtual Machine.

6 Temporal Order of Continuations

In seeking another approach to the semantics of preference, we can look to our implementation of the RVM. We recall that the prospective-value term $S \diamond E$ is both a semantic device and a programming structure within an extended language of expressions, where it can occur, for example, in an assignment such as $x := S \diamond E$. The implementation of prospective-value terms requires the calculation of E after S for each of the possible routes through S that arises from making different non-deterministic choices. Each result is added to the set of results that will comprise the value for $\{S \diamond E\}$. As this set is constructed, information concerning the order of results is being discarded, but we can retain this information by recording the values as a sequence, and this we have now implemented as the "nabla term" $S \nabla E$ where E is a sequence expression. The rules defining $S \nabla E$ are explicit about the order in which provisional choices are taken, allowing the definition of a more concrete provisional-choice operator $S \rhd T$, which tentatively executes S but will backtrack to revise its choice in favour of T if execution of S and its continuation proves infeasible.

We give two small programming examples. The first is a backtracking parser for a simple language of expressions, similar to Nelson's clairvoyant parser but avoiding left recursion (which we do by the simple expedient of using a right-associative grammar). As in Nelson's example, *Id*, *Otimes* and *Oplus* attempt to parse an identifier, a multiply symbol and an addition symbol from the input stream. If this is possible they update the input stream pointer, otherwise they enter reverse execution. T parses expressions that contain no addition symbols, and E is the complete expression parser.

$$T \ \widehat{=} \ (Id \,; Otimes \,; T) \rhd Id \quad \text{and} \quad E \ \widehat{=} \ (T \,; Oplus \,; E) \rhd T$$

As an example of a program that uses a nabla term, consider the calculation of frequencies of possible scores obtained by summing the values of three dice. We have an initialisation to record the scores associated with each of the 6^3 possible outcomes as a sequence, and an operation to interrogate the sequence and tell us the number of entries in the sequence that correspond to a given score. In this code $:\in$ represents provisional choice from a set and $\rhd$ represents Z range restriction. The initialisation code is:

$$@x_1, x_2, x_3 \bullet scores \ := \ (x_1 :\in die; \ x_2 :\in die; \ x_3 :\in die \nabla \langle x_1 + x_2 + x_3 \rangle)$$

The operation, giving the frequency of the score n, restricts the range of the sequence of scores to n and takes the cardinality of the resulting set:

$$f \leftarrow freq(n) \ \widehat{=} \ f := \text{card}(scores \rhd \{n\})$$

We turn now to the description of nabla terms. With a view to using them within both specification-level and implementation-level language we give two forms of choice. $S \rhd T$ is a provisional preferential choice which tries S before T. $S \sqcap T$ is implementor's choice, which may be resolved as a refinement step and is *not* subject to revision by backtracking. In the following rules giving the semantics of $S \nabla E$, E is a sequence expression. Implementor's choice is represented by a bunch of possible results, whilst provisional choice is sequenced to express preference.

$\mathbf{skip} \nabla E$	$=$	E	skip
$x := F \nabla E$	$=$	$E[F/x]$	assignment
$P \mid S \nabla E$	$=$	$P \,\rule{1pt}{1em}\, (S \nabla E)$	precondition
$g \rightarrow S \nabla E$	$=$	$g \twoheadrightarrow (S \nabla E), \neg g \twoheadrightarrow \langle\rangle$	guard
$S \rhd T \nabla E$	$=$	$(S \nabla E) \frown (T \nabla E)$	preferential choice
$S \sqcap T \nabla E$	$=$	$(S \nabla E), (T \nabla E)$	implementor's choice
$S \,; T \nabla E$	$=$	$S \nabla T \nabla E$	sequential composition
$@v \bullet S \nabla E$	$=$	$\oint v \bullet S \nabla E$	unbounded implementor's choice

We also have an associated refinement order $S \sqsubseteq_{\text{toc}} T \,\widehat{=}\, (T \nabla E) : (S \nabla E)$ for all sequence expressions E.

We demonstrate the rules for assignment, preference, sequential composition and guard with the following simple examples, which show how preferential choice acts in the absence and in the presence of backtracking.

$$x := 1 \rhd x := 2 \mathbin{\nabla} \langle x \rangle \ = \text{“pref choice“}$$
$$(x := 1 \mathbin{\nabla} \langle x \rangle) \frown (x := 2 \mathbin{\nabla} \langle x \rangle) \ = \text{“assignment”}$$
$$\langle 1 \rangle \frown \langle 2 \rangle \ = \ \langle 1, 2 \rangle$$

$$x := 1 \rhd x := 2 \,;\, x = 2 \rightarrow \mathbf{skip} \mathbin{\nabla} \langle x \rangle \ = \text{“sequential composition”}$$
$$x := 1 \rhd x := 2 \mathbin{\nabla} x = 2 \rightarrow \mathbf{skip} \mathbin{\nabla} \langle x \rangle \ = \text{“guard and skip”}$$
$$x := 1 \rhd x := 2 \mathbin{\nabla} x = 2 \multimap \langle x \rangle, \neg x = 2 \multimap \langle\rangle \ = \text{“pref choice”}$$
$$(x := 1 \mathbin{\nabla} x = 2 \multimap \langle x \rangle, \neg x = 2 \multimap \langle\rangle)^\frown$$
$$(x := 2 \mathbin{\nabla} x = 2 \multimap \langle x \rangle, \neg x = 2 \multimap \langle\rangle) \ = \text{“assignment”}$$
$$(1 = 2 \multimap \langle 1 \rangle, \neg 1 = 2 \multimap \langle\rangle) \frown (2 = 2 \multimap \langle 2 \rangle, \neg 2 = 2 \multimap \langle\rangle)$$
$$= \text{“properties of bunch guard”}$$

$$(\mathbf{null}, \langle\rangle) \frown (\langle 2 \rangle, \mathbf{null}) \ = \text{“bunch union with } \mathbf{null}\text{”}$$
$$\langle\rangle \frown \langle 2 \rangle \ = \ \langle 2 \rangle$$

We do not give a rule for unbounded preferential choice. We do, however, implement preferential choice from a set in the RVM, but the set must be finite, and this adds no additional expressive power over binary preferential choice. Another form of choice from a set implemented on the RVM, but beyond the scope of the current article, is to choose elements in a random order.

We no longer have the choice symbol $\Box$ in our repertoire of fundamental program connectives. Within pv semantics, choice plays the dual rôle of representing implementor's choice, to be removed during refinement, and provisional choice, to be resolved by backtracking. Since we have now teased these two concepts apart, we need an element of both in the definition that re-introduces the general notion of choice.

$$S \ \Box \ T \ \mathrel{\widehat{=}} \ (S \rhd T) \sqcap (T \rhd S)$$

We can then directly demonstrate some familiar algebraic properties of nondeterministic choice, e.g. commutativity, associativity, distribution of a guard through choice, distribution of precondition through choice, and having **magic** as a unit. However, we lose idempotence.

Let us now return to the issue of monotonicity. We recall that our first form of preferential choice, $S \ [> \ T$, resulted in a non-monotonic pv semantics. We might wonder whether the inclusion of a preferential choice must necessarily have such an effect, but, in fact, we are able to show that toc semantics *is* monotonic.

Theorem 2. ***Monotonicity of toc semantics***
For any program S and sequence expressions A and B

$$[A : B] \Rightarrow (S \mathbin{\nabla} A) : (S \mathbin{\nabla} B)$$

Proof. We prove $(S \mathbin{\nabla} A) : (S \mathbin{\nabla} B)$ under the assumption $[A : B]$. The proof is by structural induction. We have base cases for skip and assignment:

$\textbf{skip} \mathbin{\nabla} A \;=\; A$ "by toc semantics of skip".
$A : B$ "by assumption".
$B \;=\; \textbf{skip} \mathbin{\nabla} B$ "by toc semantics of skip".

and for assignment:

$x := E \mathbin{\nabla} A \;=\; A[E/x]$ "by toc semantics of assignment".
$A[E/x] : B[E/x]$ "by assumption $[A : B]$".
$B[E/x] \;=\; x := E \mathbin{\nabla} B$ "by toc semantics of assignment".

Now the inductive cases. For precondition we have:

$P \mid S \mathbin{\nabla} A \;=\; P \,\|\, (S \mathbin{\nabla} A)$ "by toc precondition rule".
Now, noting the bunch property $E : F \Rightarrow P \,\|\, E : P \,\|\, F$
$P \,\|\, (S \mathbin{\nabla} A) \;:\; P \,\|\, (S \mathbin{\nabla} B)$ "by inductive case and noted property".
$P \,\|\, (S \mathbin{\nabla} B) \;=\; P \mid S \mathbin{\nabla} B$ "toc precondition".

For guard the proof is similar to precondition but appealing to the bunch property $E : F \Rightarrow g \rightarrow E : g \rightarrow F$.

For preferential choice:

$S \rhd T \mathbin{\nabla} A \;=\; (S \mathbin{\nabla} A) \frown (T \mathbin{\nabla} A)$ "by toc pref choice".
Now noting that $E : E' \wedge F : F' \Rightarrow E \frown F : E' \frown F'$
$(S \mathbin{\nabla} A) \frown (T \mathbin{\nabla} A) \;:\; (S \mathbin{\nabla} B) \frown (T \mathbin{\nabla} B)$ "ind. case and noted property".
$(S \mathbin{\nabla} B) \frown (T \mathbin{\nabla} B) \;=\; S \rhd T \mathbin{\nabla} B$ "by toc pref choice".

For implementor's choice:

$S \sqcap T \mathbin{\nabla} A \;=\; (S \mathbin{\nabla} A), (T \mathbin{\nabla} A)$ "by toc implementor's choice".
Now noting that $E : E' \wedge F : F' \Rightarrow E, F : E', F'$
$(S \mathbin{\nabla} A), (T \mathbin{\nabla} A) \;:\; (S \mathbin{\nabla} B), (T \mathbin{\nabla} B)$ "ind. case and noted property".
$(S \mathbin{\nabla} B), (T \mathbin{\nabla} B) \;=\; S \sqcap T \mathbin{\nabla} B$ "by toc implementor's choice".

For unbounded choice:

$@v \bullet S \mathbin{\nabla} A \;=\; \oint v \bullet (S \mathbin{\nabla} A)$ "by toc unbounded choice".
Now noting that $[E : F] \Rightarrow \oint v \bullet E : \oint v \bullet F$
$\oint v \bullet (S \mathbin{\nabla} A) \;:\; \oint v \bullet (S \mathbin{\nabla} B)$ "by inductive case and noted property".
$\oint v \bullet (S \mathbin{\nabla} B) \;=\; @v \bullet S \mathbin{\nabla} B$ "by toc unbounded choice". □

Corollary 1. ***Sub-conjunctivity of toc semantics***

$$(S \mathbin{\nabla} A), (S \mathbin{\nabla} B) : (S \mathbin{\nabla} A, B)$$

Proof. Since $[A : A, B]$ we have by toc monotonicity $(S \mathbin{\nabla} A) : (S \mathbin{\nabla} A, B)$ and similarly $(S \mathbin{\nabla} B) : (S \mathbin{\nabla} A, B)$. Furthermore, we conclude by the bunch property $E_1 : F \wedge E_2 : F \Rightarrow E_1, E_2 : F$ that $(S \mathbin{\nabla} A), (S \mathbin{\nabla} B) : (S \mathbin{\nabla} A, B)$. □

Observe that toc semantics is not, however, conjunctive, and counterexamples are easy to construct, e.g. $S \mathrel{\widehat{=}} \textbf{skip} \rhd x := x + 2$, $A \mathrel{\widehat{=}} \langle x \rangle$, and $B \mathrel{\widehat{=}} \langle x + 2 \rangle$.

One further algebraic property of choice in wp and pv semantics is distribution of sequential composition through choice. Demonstration of this property requires conjunctivity; we have only sub-conjunctivity and can demonstrate only a weaker property: $S \,;\, T \;\Box\; U \sqsubseteq_{\text{toc}} (S \,;\, T) \;\Box\; (T \,;\, U)$.

To conclude, toc semantics is more discriminating than pv semantics, but should be consistent with it over the pv program connectives. Since toc semantics captures additional information about the order in which results are produced, but describes the same results as pv semantics, we require that $S \nabla \langle E \rangle$ should produce a sequence whose range is equal to $S \diamond E$. This is easily proved as a corollary of the following more general theorem.

Theorem 3. ***Consistency of pv and toc semantics***
For any program S defined over the connectives described by pv semantics, and for any sequence expression E, we have $S \diamond {\sim}\text{ran}(E) = {\sim}\text{ran}(S \nabla E)$.

Proof. The proof is by structural induction, once again with base cases for skip and assignment:

$$\begin{aligned}
&\textbf{skip} \diamond {\sim}\text{ran}(E) \;=\; \text{“pv semantics of skip”}\\
&{\sim}\text{ran}(E) \;=\; \text{“toc semantics of skip”}\\
&{\sim}\text{ran}(\textbf{skip} \nabla E)
\end{aligned}$$

and for assignment:

$$\begin{aligned}
&x := F \diamond {\sim}\text{ran}(E) \;=\; \text{“pv semantics of assignment”}\\
&{\sim}\text{ran}(E)[F/x] \;=\; \text{“property of substitution”}\\
&{\sim}\text{ran}(E[F/x]) \;=\; \text{“toc semantics of assignment”}\\
&{\sim}\text{ran}(x := F \nabla E)
\end{aligned}$$

For the inductive cases we provide proofs only for some example language structures. We choose guard, choice and sequential composition. For guard we have:

$$\begin{aligned}
&g \rightarrow S \diamond {\sim}\text{ran}(E) \;=\; \text{“pv semantics of guard”}\\
&g \twoheadrightarrow (S \diamond {\sim}\text{ran}(E)) \;=\; \text{“by appeal to inductive case”}\\
&g \twoheadrightarrow {\sim}\text{ran}(S \nabla E) \;=\; \text{“by case analysis on } g\text{”}\\
&{\sim}\text{ran}(g \twoheadrightarrow (S \nabla E)) \;=\; \text{“by toc semantics of guard”}\\
&{\sim}\text{ran}(g \rightarrow S \nabla E)
\end{aligned}$$

For choice we have:

$$\begin{aligned}
&S \;\Box\; T \diamond {\sim}\text{ran}(E) \;=\; \text{“pv semantics of choice”}\\
&(S \diamond {\sim}\text{ran}(E)), (T \diamond {\sim}\text{ran}(E)) \;=\; \text{“by appeal to inductive case”}\\
&{\sim}\text{ran}(S \nabla E), {\sim}\text{ran}(T \nabla E) \;=\; \text{“by property } {\sim}A, {\sim}B = {\sim}(A \cup B)\text{”}\\
&{\sim}(\text{ran}(S \nabla E) \cup \text{ran}(T \nabla E)) \;=\; \text{“by law ran}(s) \cup \text{ran}(t) = \text{ran}(s \frown t)\text{”}\\
&{\sim}\text{ran}((S \nabla E) \frown (T \nabla E)) \;=\\
&\text{“by property ran}(s \frown t) = \text{ran}(t \frown s) \text{ and idempotence of bunch union”}\\
&{\sim}(\text{ran}((S \nabla E) \frown (T \nabla E), (T \nabla E) \frown (S \nabla E))) \;=\; \text{“by toc pref choice”}\\
&{\sim}\text{ran}((S \rhd T \nabla E), (T \rhd S \nabla E)) \;=\; \text{“by toc defn of } S \;\Box\; T\text{”}\\
&{\sim}\text{ran}(S \;\Box\; T)
\end{aligned}$$

For sequential composition:

$$\begin{aligned}
&S\,;T \diamond {\sim}\mathrm{ran}(E) \;=\; \text{"pv semantics of sequential composition"}\\
&S \diamond T \diamond {\sim}\mathrm{ran}(E) \;=\; \text{"inductive case"}\\
&S \diamond {\sim}\mathrm{ran}(T \nabla E) \;=\; \text{"inductive case"}\\
&{\sim}\mathrm{ran}(S \nabla T \nabla E) \;=\; \text{"toc semantics of sequential composition"}\\
&{\sim}\mathrm{ran}(S\,;T \nabla E)
\end{aligned}$$

□

7 Conclusions

We have described our search for a method to express preference in the context of non-deterministic choice. We first considered Nelson's biased choice, but found it was too short-sighted to meet our needs. We then looked to probabilistic choice for inspiration, and we constructed a form of preferential choice. However, on inspection we found this made our calculus non-monotonic. Although this opens interesting perspectives, the loss of monotonicity is not attractive. Finally, we looked for inspiration to our Reversible Virtual Machine. This has a programming structure that will collect all the possible results of a non-deterministic computation. We added a similar structure which records the results of a search as a sequence. The formal description of this structure forms the basis for a calculus which captures preference by representing provisional choice in terms of sequences of possible expression values to be passed to a continuation. By adding separate formulations for implementor's choice, we obtain descriptions for the essential programming connectives of a reversible guarded command language with preferential choice. Since provisional choice is now ordered and the rôle of continuations is paramount, we call this a "temporal order of continuations" semantics. We give a refinement ordering which allows implementor's choice to be reduced and preconditions to be widened. We showed that toc semantics is monotonic, though only sub-conjunctive, and we established that it is consistent with prospective-value semantics.

Future discussions will consider the partial-order properties of the toc refinement relation and its use in fixed-point treatments. Additional items on the agenda are the description of an associated relational model, the investigation of a probabilistic unification, and elaboration of the proof obligations required to employ toc as a refinement-based development method.

Acknowledgements. We acknowledge the helpful comments of the referees and thank Walter Guttmann for an interesting electronic correspondence.

References

1. Bennett, C.H.: Logical Reversibility of Computation. IBM Journal of Research and Development 17(6), 525–532 (1973)
2. Dunne, S.E.: Chorus Angelorum. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 19–33. Springer, Heidelberg (2006)

3. Feynman, R.P.: Lectures on Computation. Westview Press (1996)
4. Floyd, R.W.: Nondeterministic Algorithms. J. of the ACM 14(4), 636–664 (1967)
5. Hehner, E.C.R.: A Practical Theory of Programming. Springer, Heidelberg (1993), Latest version available online at: `http://www.cs.toronto.edu/~hehner/aPToP/`
6. Landauer, R.: Irreversibility and Heat Generated in the Computing Process. IBM Journal of Research and Development 5, 183–191 (1961)
7. Morris, J.M., Bunkenburg, A.: A Theory of Bunches. Acta Informatica 37(8), 541–561 (2001)
8. Nelson, G.: A Generalization of Dijkstra's Calculus. ACM Transactions on Programming Languages and Systems 11(4), 517–561 (1989)
9. Stoddart, W.J., Zeyda, F.: A unification of probabilistic choice within a design-based model of reversible computation. Formal Aspect of Computing (August 2007) (Published on-line), doi:10.1007/s00165-007-0048-1
10. Stoddart, W.J., Zeyda, F., Lynas, A.R.: A Design-based model of reversible computation. In: Dunne, S., Stoddart, B. (eds.) UTP 2006. LNCS, vol. 4010, pp. 63–83. Springer, Heidelberg (2006)
11. Stoddart, W.J., Zeyda, F., Lynas, A.R.: A reversible virtual machine. In: Proceedings of Reversible Computation 2009 (March 2009)
12. Stoddart, W.J., Zeyda, F., Lynas, A.R.: A Virtual Machine for Supporting Reversible Probabilistic Guarded Command Languages. Electronic Notes in Theoretical Computer Science 253(6), 33–56 (2010)
13. Zeyda, F.: Reversible Computations in B. PhD thesis, University of Teesside, Middlesbrough, TS1 3BA, UK (July 2007)
14. Zuliani, P.: Logical reversibility. IBM Journal of Research and Development 45(6), 807–818 (2001)

Material Flow Abstraction of Manufacturing Systems*

Jewgenij Botaschanjan and Benjamin Hummel

Institut für Informatik, Technische Universität München, Germany
{botascha,hummelb}@in.tum.de

Abstract. Manufacturing systems integrate electro-mechanical components with software to fulfill certain production tasks. Due to this combination of computer science and traditional engineering, formal models of embedded systems are often inappropriate for the description and analysis of manufacturing systems. This is especially prominent on an abstract level, where computation and communication are not of primary concern, but rather the material the system is processing. In this paper, we introduce an abstract formal model of manufacturing systems based on the material flow, i.e., the relation between incoming and outgoing material over time. The formalization supports compositional reasoning and the comparison of specifications with more concrete models (implementations). This provides a foundation for the formally founded conceptual modeling of manufacturing systems and the reasoning about the correctness of their realization. The former is evaluated by a prototypical tool implementation and a case study.

1 Introduction

Manufacturing systems are defined as systems for producing certain goods by transportation, assembly, and modification of input material. Usually, this is limited to discrete products[1], which we also assume in this paper. As the mode of operation depends on both electro-mechanic effects and software, these systems belong to the larger class of mechatronic or cyber-physical systems. The size of these systems ranges from smaller ones (e.g., for mounting of circuit boards) to systems filling entire factory buildings (such as the assembly lines in car manufacturing). All of these systems typically consist of a collection of machine tools, automation and transportation systems, and industrial robots, which interact with each other to solve a certain production problem.

As in similar domains, competition and price erosion lead to increasingly complex systems, which complicate the application of traditional engineering and development processes. One reason for this is the lack of a common behavior model describing the entire machine (actors/sensors and software). Especially, assuring functional correctness in all circumstances, which also includes safe operation in the case of errors, is nearly impossible without an underlying formal behavior model and suitable tool support. While first steps towards a model for manufacturing systems have been taken, both from a theoretic [1] and a more practical perspective [2], these approaches do not

* This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant "SPES2020, 01IS08045A".

[1] Dealing with liquids is usually summarized by the term *process technology*.

directly facilitate the conceptual modeling but rather provide a programming model for such systems. The ability to formally express system properties and to reason about their satisfaction is the second obligatory ingredient to ensure functional correctness.

For manufacturing systems a natural property or conceptual view is the reduction of a subsystem to its material flow. For this, only the relation between incoming and outgoing elements of material over time is considered. Thus, only the material, which is at the core of the system's task, is included in the model, while communication between subsystems or positions of machine parts are neglected. This approach moves the focus of the models more to the requirements phase, as the actual solution of a transportation or modification problem in terms of mechanical parts and software is left unspecified.

Problem Statement. While there are many models for mechatronic and manufacturing systems, most of them either do not capture the behavior (such as CAD models) or are very detailed (often based on physics simulation) and only a small number of them is built upon a formal mathematical framework, making analysis and automatic reasoning hard, if not impossible (c.f., Sec. 7). Especially for highly abstract, still generally applicable models of manufacturing systems there is no rigorous mathematical theory. We consider this situation precarious, as it hampers the development of novel techniques and tools for the development and verification of such systems.

Contribution. This paper provides a theory of material flow interfaces (MFIs), which describe components of a manufacturing system in terms of material exchanged at their boundaries (Sec. 3). The modeling theory is a first step towards a very abstract, but formal description technique for these systems. We discuss important properties of our model and describe an operator for constructing new MFIs by composition of others (Sec. 4). Additionally, conformance of an implementation of an MFI (Sec. 2) to the abstract specification is defined in Sec. 5. Finally, in Sec. 6 we provide prototypical tool support for MFI based specifications and report on a first small case study.

2 Modeling of Manufacturing Systems

Here, we present an implementation model of manufacturing systems which consists of communicating components, whose actions are interpreted in a physical environment: dedicated outputs are made to movements of certain volumes (e.g., material or machine parts), dedicated inputs represent collisions in space (e.g., between material and a light barrier). This is a natural programming model for the system class considered (c.f., [1]).

We model material as objects which have a *volume* and a *state*. Volume describes both shape and position of material, while state can be used to uniquely identify material pieces or encode their status in the workflow. We assume a set of possible volumes V and states S for material objects. The term *volume* here means a subset of $\mathbb{R}^3$ which encodes both shape and position and which is closed under union, i.e., for all $v_1, v_2 \in V$ holds $v_1 \cup v_2 \in V$. At any given time, a material object is uniquely determined by a pair from $V \times S$.

A *material configuration* is a finite set $C \subset V \times S$. The set of all material configurations is denoted by $\mathcal{MC} \subset \mathcal{P}(V \times S)$. A *material transformation* is a function

$\mathcal{MC} \to \mathcal{MC}$ which changes both the position/shape and state for all material objects in the (observed) world. The set of all possible material transformation functions is named $\mathcal{MT}$. This definition of a material transformation also captures the cases where material objects are split or joined or are completely consumed (for example in a combustion chamber). $\mathcal{RT}$, a subset of $\mathcal{MT}$, contains all shape preserving and state invariant transformations, i.e., (combinations of) rotations and translations.

For modeling time, we follow the stream-based approach described in [3]. Let $\mathcal{M}$ be an arbitrary set and $\mathbb{T}$ the time domain (e.g., $\mathbb{N}$ or $\mathbb{R}_{\geq 0}$), then $\bigcup_{i \in \mathbb{T}}([0, i) \to \mathcal{P}(\mathcal{M}))$ models finite streams and $\mathbb{T} \to \mathcal{P}(\mathcal{M})$ infinite ones. By $\mathcal{M}^{\mathbb{T}}$ we denote the set of all infinite streams over $\mathcal{M}$. For $\sigma \in \mathcal{M}^{\mathbb{T}}$ and $t \in \mathbb{T}$ we write $\sigma.t \subseteq \mathcal{M}$ for elements of stream σ at time t. For a function $f\colon X \to Y$ by $f|_D$ we denote the *restriction* of the domain of f to D, i.e., $f|_D\colon (X \cap D) \to Y$ and for all $d \in (X \cap D)$ holds $f|_D(d) = f(d)$. Then, the finite prefix σ until time t is written as $\sigma{\downarrow_t}$ and defined as $\sigma|_{[0,t]}$, i.e., the restriction of the function to the closed time interval until t.

A stream $f \in (\mathcal{MT})^{\mathbb{T}}$ is called a material flow if it is not *dense*, i.e., for any $t \in \mathbb{T}$ the set $\{s \in \mathbb{T} \mid s < t \ \wedge \ f(s) \neq \mathrm{id}\}$ is finite, where id is the identity function, – every time there were finitely many valuation changes in f. For material of configuration $C \in \mathcal{P}(V \times S)$ their positions and locations at time $t \in \mathbb{T}$ for a given material flow f are determined by $f_n(f_{n-1}(\ldots f_1(f_0(C))\ldots))$, where $f_0, \ldots, f_n$ is the ordered sequence of material transformations determined by picking the (finitely many) non-id material transformations from f until time t. Thus, motion of material over time is described by a number of discrete changes to both the volume (position and shape) and state. There are some properties which can be required for a material flow to be *valid*, such as to never cause material volumes to overlap or to preserve a notion of continuity. As we do not require these properties in this paper, we will not discuss them further.

With these ingredients we represent an *abstract manufacturing system (AMS)* by two sets I_C and O representing (typed) input and output variables, a set I_S representing sensor input, a volume N describing its interior, a function $S : \mathcal{MC} \to (I_S \to \mathrm{type}(I_S))$ describing variables corresponding to sensors, as well as a behavior function $F : \overrightarrow{(I_C \cup I_S)} \to \mathcal{P}(\vec{O} \times \mathcal{MF})$, where $\overrightarrow{(I_C \cup I_S)}$ and $\vec{O}$ denote the valuations of the variables over time (i.e., $\vec{X} \overset{\mathsf{def}}{=} \{X \to (\mathrm{type}(X))^{\mathbb{T}}\}$) and $\mathcal{MF} \subseteq (\mathcal{MT})^{\mathbb{T}}$ denotes the set of all valid material flows. The sets I_C and O allow communication between different components (manufacturing systems), while I_S represents inputs from sensors measuring properties of material objects and is uniquely determined by the current material configuration via the sensor function S. Thus, I_C is affected directly by other components or user input, while I_S is determined by the environment (material configuration) which can only be influenced indirectly. We assume sensors to be state-less and thus S is not a function on streams, but on values at a single time which is evaluated for any point in time. The power set for the function's results allows for non-deterministic behavior. The volume N describes the *interior* of the system, which is the space in which the system can affect material. The possible material flows describe the effect of the manufacturing system and its actuators to the material being within or touching the interior of the AMS.

In this definition of an AMS all possible outputs and material flows for a complete input history are provided by the behavior function. For an AMS to be plausible we thus have to require the behavior function F to be *strongly time guarded*, i.e., there has to be a $\delta > 0$ such that for any two input histories $i_1, i_2 \in \vec{I}$ and time $t \in \mathbb{T}$

$$i_1{\downarrow_t} = i_2{\downarrow_t} \implies \{(o,f){\downarrow_{t+\delta}} \mid (o,f) \in F(i_1)\} = \{(o,f){\downarrow_{t+\delta}} \mid (o,f) \in F(i_2)\} ,$$

where the prefix operator is applied point wise. This equation captures the intuition that any event may only depend on inputs observed in the past, and that there is a processing delay between inputs and outputs (of at least δ).

We will not discuss composition of AMS in detail, as the focus of this paper are material flow abstractions. The overall idea for composition is to *connect* the inputs and outputs of AMS components, thus ensuring equality of their communication histories. This leads to a system of equations, which has a unique solution thanks to strong time guardedness (c.f., [4]). The possible material flows of the composed system are restricted by the possible material flows of both composed AMS components, thus they are defined by their intersection.

3 Material Flow Interfaces

This section describes the abstracted model reducing an AMS to its material flow. On a conceptual level a manufacturing system can be seen as a transformer function on material streams. It expects certain pieces of material, e.g., rough parts of certain shape and size, and outputs modified material, e.g., pieces welded together, over time. At that point of view a transportation system is a buffer, which changes the spatial position and orientation of incoming pieces and a milling machine reduces the volume of the material and alters its production level state.[2] All these processes must be considered over time to take the throughput of the respective system into account. We call the interface specification of incoming and outgoing material pieces over time *material flow interface* (*MFI*).

With this model in mind, we can model more complex systems by combining their MFIs. This ensures that the assumptions they put on the incoming material flow will fit together with the outgoing flows they produce (guarantee). For example, the transportation system must supply a machine tool with rough parts which it can process and at a rate which corresponds to its processing rate. Analogous to the refinement in compositional frameworks for software systems, provided the systems fit together (are composable) on the level of MFI, the respective "implementations", i.e., the actual manufacturing components, are guaranteed to fit together as well.

An MFI consists of a number of *entry* and *exit interface points*. They describe places in space and time where the material enters/leaves the system, resp. They also specify the kind of material expected there. In other words, every point is specified by a spatial body and material flows, which are expected to be observed within that body. For example, consider a milling machine which expects aluminium cuboids with minimal inter-arrival time of 75s at its forehold and issues a motor body for every cuboid after 64s at the same place. This can be modeled by an MFI consisting of one entry and one exit both placed at the forehold with associated material flows as described above.

[2] A modern machine tool manages information about its working pieces and reports, e.g., on success of their processing, to the outer world.

The placement of the interface points depends in the first place on the manufacturing process to be modeled. There exist an unbounded number of places where material can be put on/taken away from a conveyor. However, a certain technological process could restrict this possibility to exactly one entry and exactly one exit point. We consider any further material, which enters/exits a system *not* through its interface points as illegal input to/output of the system, resp. The process of entering/exiting a system is modeled as an intersection between material volumes and an interface point.

Analogous to the I/O ports in models of software, we can establish a material flow between systems by connecting an exit/entry pair together. This can be done when volumes of entry and exit intersect. However, there exist substantial differences between data and material flow which have to be taken into account: (1) A message is transferred at once (or with a certain delay); a piece of material can pertain to and be affected by both issuing and receiving subsystem for a certain time. (2) A message can arrive to a system through its input ports only; a piece of material can also appear in the system in an unexpected position. (3) Channel hiding does directly work for interface points – every material exchange can be interfered by external events. (4) Messages can be broadcasted, material cannot be duplicated. But, material can be passed on to the same system by several predecessors in the manufacturing chain and several systems can obtain material from the same predecessor. These observations yield a number of properties a realistic MFI must exhibit (c.f., Sec. 3.2) and influence the definition of MFI (Sec. 3.1) and MFI combination (Sec. 4). Next, we give a formal foundation of MFI.

3.1 Formal Definition

First, we need to clarify the intuition behind a material stream. In contrast to AMS, in MFIs a stream does *not* incorporate the trajectories of material pieces through time and space but resembles the notion of *observation* from the software world. This difference is illustrated by Fig. 1: its upper part shows trajectories of three material pieces on a conveyor. The conveyor moves the material at a constant speed, halts for a moment, and proceeds subsequently with the same speed. This concrete behavior is irrelevant when considering the conveyor as a material processing function – only the points in time and

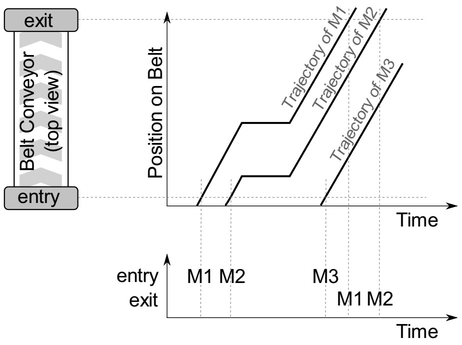


Fig. 1. Relation between Material Trajectories (top) and MFI Streams (bottom)

space of entering and exiting the conveyor are of importance. Thus, the streams of the conveyor's MFI specification, shown in the lower part of Fig. 1, are the *projections* of material trajectories to observation on conveyor's entry and exit.

An MFI stream $\sigma \in (V \times S)^{\mathbb{T}}$ documents times and positions of the *initial* observations of material pieces at some interface point. $mo \in \sigma.t$ means that mo has initially appeared in that interface point at time t. Since the appearance of mo is a physical rather than logical process, it can also remain in the point's volume for a certain time. However, this fact is neglected by our semantic model. Thus, $mo \in \sigma.t'$ for some $t' \neq t$ is interpreted as the occurrence of *another* material instance.

Next, we give a formal definition of the MFI tuple. Its main difference to a model of software is the description of syntactic interface. It consists not only of a type function but also of two volume functions: vol and $novol$. These functions define the allowed and forbidden areas of material observation, resp. In particular, setting vol to an empty set means that no observation is allowed. At first view, $novol$ may seem to be redundant if we make a closed world assumption – everything that is not allowed is forbidden. However, in the settings of spatial environment the *open world assumption* (*OWA*) is more feasible since such an environment can interfere any time at any place – the knowledge of an MFI about its environment is incomplete. Under OWA we need to document areas where environment may not interfere explicitly.

Definition 1. *An MFI is defined as a material flow processing function. Formally, we write MFI* $\stackrel{\text{def}}{=} (EN, EX, vol, val, novol, ass, gar)$*, where* $EN = \{en_1, \ldots, en_n\}$ *and* $EX = \{ex_1, \ldots, ex_m\}$ *are disjoint sets of entries and exits. By IP* $\stackrel{\text{def}}{=} EN \cup EX$ *we denote the overall syntactical material flow interface.*

Every interface point $ip \in IP$ *is mapped to a volume over time by* $vol(ip) \in \mathbb{T} \rightarrow V$*. These are areas where material pieces incoming to/outgoing from* ip *are expected to be observed. We demand that for all* $t \in \mathbb{T}$ *no pair of interface points intersects:* $\forall ip_1, ip_2 \in IP : ip_1 \neq ip_2 \implies vol(ip_1).t \cap vol(ip_2).t = \emptyset$.

The type of material which may flow through an interface point is fixed by $val(ip) \subseteq (V \times S)$*. We demand that* val *is closed under shape-preserving transformations, i.e., for all* $(v, s) \in val(ip)$ *and* $r \in \mathcal{RT}$ *must hold* $r(v, s) \in val(ip)$.

The $novol$ *function with signature* $\mathbb{T} \rightarrow V$ *describes* forbidden *volumes of the MFI. Nothing may be observed there. By this,* $novol$ *realizes "information hiding" in space. There may be no intersection between* $novol$ *and* vol *any time for any interface point.*

Each interface point $ip \in IP$ *is associated with a set of material streams* $\overrightarrow{ip} \subseteq (V \times S)^{\mathbb{T}}$*, such that for all* $\sigma \in \overrightarrow{ip}$ *at any time* $t \in \mathbb{T}$ *a piece of material from* $val(ip)$ *intersects with the volume* $vol(ip).t$ *and does* not *intersect with* $novol.t$*. Formally,* $\sigma \in \overrightarrow{ip}$ *iff for all* $t \in \mathbb{T}$, $(s, v) \in (V \times S)$ *holds*

$$(s, v) \in \sigma.t \Leftrightarrow (s, v) \in val(ip) \wedge (v \cap vol(ip).t) \neq \emptyset \wedge (v \cap novol.t) = \emptyset \ .$$

$\overrightarrow{IP}$ *denotes the valuation set of interface points from IP.*

The predicate $ass \colon \overrightarrow{EN} \rightarrow \mathbb{B}$ *captures the assumption about incoming material streams, while* $gar \colon \overrightarrow{EN} \rightarrow (\overrightarrow{EX} \rightarrow \mathbb{B})$ *is the guarantee about the material streams produced by MFI. For a stream* $\sigma \in \overrightarrow{IP}$ *we abbreviate* $ass(\sigma) \stackrel{\text{def}}{=} ass(\sigma|_{EN})$ *and* $gar(\sigma) \stackrel{\text{def}}{=} gar(\sigma|_{EN})(\sigma|_{EX})$. □

The overall behavior of MFI contains streams over IP which satisfy gar under assumption ass. The set of streams described by MFI is denoted by $[\![MFI]\!]$ defined as

$$[\![MFI]\!] \overset{\text{def}}{=} \{\sigma \in \overrightarrow{IP} \mid ass(\sigma) \wedge gar(\sigma)\} \ . \tag{1}$$

We expect MFIs to be strongly time guarded (c.f., Sec. 2), i.e., every process step must require time. This ensures the existence of a unique fixed point of every computation described by a (composite) MFI [3].

A special feature of our model is that the enabling behavior of interface points is specified by the vol function separately from the actual MFI behavior described by ass/gar. The time dependent and input independent function vol provides an over-approximation about the observable spatial behavior on the system bounds. This makes spatial dependencies between MFI models explicit, so we can reason about combinability of MFIs (c.f., Sec. 4) on a behavior independent level.

Example 1. We define an MFI specification of the milling machine described above by $MFI_{mm} \overset{\text{def}}{=} (\{en_{mm}\}, \{ex_{mm}\}, vol, val, \mathbb{T} \to \emptyset, ass, gar)$. It processes one piece at a time at the rate of 75s and for every input rough part issues exactly one part, which may be OK or waste, after 64s. Formally,

$$\forall t \in \mathbb{T} : vol(en_{mm}).t \overset{\text{def}}{=} \begin{cases} v_{forehold} & \text{if } \exists k \in \mathbb{N} : t = 75 * k, \\ \emptyset, & \text{otherwise,} \end{cases}$$

$$\forall t \in \mathbb{T} : vol(ex_{mm}).t \overset{\text{def}}{=} \begin{cases} v_{forehold} & \text{if } \exists k \in \mathbb{N} : t = 75 * k + 64, \\ \emptyset, & \text{otherwise,} \end{cases}$$

$$val(en_{mm}) \overset{\text{def}}{=} \{(v_{rough}, rough)\}, \quad val(ex_{mm}) \overset{\text{def}}{=} \{v_{part}\} \times \{OK, fail\},$$

$$\forall \sigma \in \overrightarrow{en}_{mm} : ass(\sigma) \overset{\text{def}}{=} \forall t \in \mathbb{T} : |\sigma.t| \leq 1,$$

$$\forall \sigma \in \overrightarrow{IP} : gar(\sigma) \overset{\text{def}}{=} \forall t \in \mathbb{T} : |\sigma(en_{mm}).t| = |\sigma(ex_{mm}).(t + 64)| \ .$$

We observe that, although located at the same position, the interface points never intersect and that the assumption states that the machine accepts at most one piece at a time. Pieces can only enter the machine at the multiple of 75s because of the vol-predicate, which maps the entry to an empty set any other time. Then, MFI_{mm} guarantees that if a rough part has entered, exactly one machined part will exit after 64s; nothing will happen, otherwise. □

3.2 Important Properties of MFI

The ability of our MFI models to abstract from the implementation details comes along with the possibility to specify unrealistic behavior. Consider an MFI which produces material pieces which intersect in space. In order to exclude such behavior we formulate a number of well-formedness conditions which MFI specifications must fulfill to be realizable.

Rigid objects occupy a certain non-zero volume and may not intersect. A stream σ is called *spatially plausible* if it consists of non-intersecting material pieces only.

Formally, it must hold $\forall t \in \mathbb{T} : \forall (v_1, s_1), (v_2, s_2) \in \sigma.t : (v_1, s_1) \neq (v_2, s_2) \Rightarrow v_1 \cap v_2 = \emptyset$. An MFI is spatially plausible if for all spatially plausible entry streams it produces spatially plausible exit streams.

Only a finite number of rigid objects can be observed within finite time windows, thus, we say that a stream σ is *sparse* if for all $t \in \mathbb{T}$ the set $\{(t', mo) \mid mo \in \sigma.t' \wedge t' < t\}$ is finite. An MFI is sparse if for any sparse input stream it produces only sparse output ones.

In order to be able to arrange subcomponents specified by MFIs in space, we expect them to be *movable*, i.e., invariant resp. shape-preserving transformations, like translation and rotation. For $\mathit{MFI} = (\mathit{EN}, \mathit{EX}, \mathit{vol}, \mathit{val}, \mathit{novol}, \mathit{ass}, \mathit{gar})$ and $(\tau, \mathrm{id}) \in \mathcal{RT}$ we define $\tau(\mathit{MFI}) \overset{\text{def}}{=} (\mathit{EN}, \mathit{EX}, \tau \circ \mathit{vol}, \mathit{val}, \tau \circ \mathit{novol}, \mathit{ass}, \mathit{gar})$. It must hold $[\![\mathit{MFI}]\!] = \tau^{-1}([\![\tau(\mathit{MFI})]\!])$, where $\tau^{-1}(.)$ denotes the point-wise application of the inverse transformation (τ^{-1}, id) on every material piece from MFI's streams. This property naturally holds for all MFIs, which do not refer to absolute positions of material in their ass- and gar-predicates.

We can easily show that the milling machine MFI form Example 1 is physically plausible, sparse, and movable.

Example 2. We specify a bidirectional transportation system for handling the material exchange with the milling machine: $\mathit{MFI}_{tr} \overset{\text{def}}{=} (\{en_1, en_2\}, \{ex_1, ex_2\}, \mathit{vol}, \mathit{val}, \mathbb{T} \to \emptyset, \mathit{ass}, \mathit{gar})$. It describes both transport directions: en_1, ex_1 deliver the rough material from the environment to the machine and en_2, ex_2 return machined parts back. The transporting delay in both directions is expected to be 5s. Formally, let en_{mm}, ex_{mm} be interface points of the milling machine as introduced by Example 1, then

$$\forall t \in \mathbb{T} : \mathit{vol}(ex_1).t \overset{\text{def}}{=} \mathit{vol}_{mm}(en_{mm}).t \wedge \mathit{vol}(en_2).t \overset{\text{def}}{=} \mathit{vol}_{mm}(ex_{mm}).t,$$

$$\forall t \in \mathbb{T} : \mathit{vol}(en_1) = v_{in} \wedge \mathit{vol}(ex_2) = v_{out},$$

$$\mathit{val}(en_1) = \mathit{val}(ex_1) \overset{\text{def}}{=} \mathit{val}_{mm}(en_{mm}), \quad \mathit{val}(en_2) = \mathit{val}(ex_2) \overset{\text{def}}{=} \mathit{val}_{mm}(ex_{mm}),$$

$$\forall \sigma \in \overrightarrow{\{en_1, en_2\}} : \mathit{ass}(\sigma) \overset{\text{def}}{=} \text{true},$$

$$\forall \sigma \in \overrightarrow{\mathit{IP}} : \mathit{gar}(\sigma) \overset{\text{def}}{=} \forall t \in \mathbb{T} : \sigma(en_1).t = r_{tr}(\sigma(ex_1).(t+5)) \wedge \sigma(en_2).t = r_{tr}^{-1}(\sigma(ex_2).(t+5)),$$

where $r_{tr} \in \mathcal{RT}$ is a shape-preserving transformation and v_{in}, v_{out}, $v_{forehold}$ are pairwise disjoint. The first conjunct of the guarantee deals with the flow to and the second – from the machine, resp. They both demand any incoming material object to exit after 5s and permit only those outgoing objects which entered 5s earlier, i.e., no "spontaneous outputs" are allowed. Due to our semantics definition (Eq. (1)) MFI_{tr} obtains material on en_2 at $75 * k + 64$ only ($k \in \mathbb{N}$), although it may get more than one piece at a time. The outputs on ex_1 are only made at multiples of 75s.

We observe that MFI_{tr} is strongly time guarded (this is ensured by transporting time > 0), sparse (for one input exactly one output is made), and movable, but it is not spatially plausible. This can be fixed by adding $\forall (v_1, s_1), (v_2, s_2) \in \sigma(ex_i).t : v_1 \cap v_2 = \emptyset$ as a further conjunct to gar. We denote the modified guarantee by gar' and MFI by MFI'_{tr}. $\square$

4 Composition of MFIs

We compose MFIs by intersecting their entries and exits. However, a connected pair does not disappear from the combined interface, since an entry can be used by several exits and vice versa. Think of two robots, which are located at opposite sides of a conveyor, and interchangeably load and unload material. On the other side, we cannot allow the same piece of material to be consumed by more than one receiver at a time, since this is not physically plausible. Thus, non-intersecting volume parts of interface points remain in the composite interface, while the intersecting ones disappear.

Let there be a pair of MFIs $MFI_i = (EN_i, EX_i, vol_i, val_i, novol_i, ass_i, gar_i)$ with $i \in \{0,1\}$. A pair of interface points $ip_0 \in IP_0, ip_1 \in IP_1$ is called *intersecting* iff there exists a point in time $t \in \mathbb{T}$ such that $vol_0(ip_0).t \cap vol_1(ip_1).t \neq \emptyset$. We write $meet(ip_0, ip_1)$ to denote that ip_0 and ip_1 intersect as described above. By definition it holds that $meet$ is commutative and $meet(ip_0, ip_1)$ only if ip_0, ip_1 do not belong to the same MFI.

Definition 2. *We call MFI_0, MFI_1* composable *if they have no homonymous interface points, i.e., $IP_0 \cap IP_1 = \emptyset$, and for every pair of intersecting interface points $ip_0 \in IP_0, ip_1 \in IP_1$ (i.e., $meet(ip_0, ip_1)$) following conditions hold: (1) they are not unidirectional, i.e., ip_0, ip_1 cannot be both entries or both exits, (2) they are of the same type, i.e., $val_0(ip_0) = val_1(ip_1)$, and (3) they do not intersect with forbidden areas, i.e., for all $i \in \{0,1\}$, $t \in \mathbb{T}$, $ip \in IP_i$ holds $vol_i(ip).t \cap novol_{1-i}.t = \emptyset$.* □

For purposes of reuse and simultaneous interface design *optimistic composition* [5] seems to be more natural. It is defined even when there are streams which do not satisfy the implications between assumptions and guarantees of components being composed. Thus, the semantics of composition consists of all satisfying streams. In the composition we quantify the observations at interface point intersections existentially and reduce the composite interface by these common volumes.

We denote the composition by $MFI_0 \otimes MFI_1 \overset{\text{def}}{=} (EN_0 \cup EN_1, EX_0 \cup EX_1, vol, val_0 \cup val_1, novol, ass, gar)$, where vol is defined for all $i \in \{0,1\}$ as

$$\forall ip \in IP_i, t \in \mathbb{T} : vol(ip).t \overset{\text{def}}{=} vol_i(ip).t \setminus \bigcup_{ip' \in IP_{1-i}} vol_{1-i}(ip').t \ . \tag{2}$$

By this construction, we obtain an interface without common volumes and, since $\overrightarrow{IP}$ defines the domain and the range of A/G predicates, no inputs are accepted or outputs are made at these areas by the composite MFI.

To ensure that nothing in the environment, e.g., no further MFI, will take or release material within the common volumes of MFI_0, MFI_1 these volumes are captured by $novol$. This ensures the realizability of MFI specifications and also the associativity of our composition. Formally, for all $t \in \mathbb{T}$

$$novol.t \overset{\text{def}}{=} novol_0.t \cup novol_1.t \cup \bigcup_{ip \in IP_0, ip' \in IP_1} vol_0(ip).t \cap vol_1(ip').t \ . \tag{3}$$

In order to define the composite semantics, we introduce the following auxiliary operations on streams. Set-theoretic operations on $\sigma_0, \sigma_1 \in (V \times S)^{\mathbb{T}}$ are defined as their application along the time line, i.e., $(\sigma_0 \sim \sigma_1).t \overset{\text{def}}{=} \sigma_0.t \sim \sigma_1.t$ for all $\sim \in \{\cup, \cap, \setminus\}$ and $t \in \mathbb{T}$. For a stream σ and a volume function $vl\colon \mathbb{T} \to V$ we define a *filtered* stream $vl©\sigma$ by $\forall t \in \mathbb{T} : (vl©\sigma).t \overset{\text{def}}{=} \{(v, s) \in \sigma.t \mid v \cap vl.t = \emptyset\}$, i.e., all material objects, which intersect with vl, are filtered out. We extend the above operations for streams from $\overrightarrow{IP}$ by applying them for every interface point.

For a given stream $\sigma \in \overrightarrow{IP}$ we describe its *valid projections* $\sigma_0 \in \overrightarrow{IP}_0$, $\sigma_1 \in \overrightarrow{IP}_1$ by the proposition

$$\sigma = novol©(\sigma_0 \cup \sigma_1) \wedge \bigcup_{ip \in IP} (\sigma_0 \setminus novol©\sigma_0)(ip) = \bigcup_{ip \in IP} (\sigma_1 \setminus novol©\sigma_1)(ip),$$

i.e., the composite behavior contains both sub-behaviors in non-forbidden areas and the behavior in forbidden areas of both sub-services coincides. This corresponds to the existential quantification for valuations of internal channels which is used to describe parallel composition of software systems.

Definition 3. *For composable MFIs MFI_0, MFI_1 as above the semantics of $MFI_0 \otimes MFI_1$ is given by ass/gar-predicates defined as follows. We define*

$$ass(\sigma) \overset{\text{def}}{\Leftrightarrow} ass_0(\sigma_0) \wedge ass_1(\sigma_1) \quad \text{and} \quad gar(\sigma) \overset{\text{def}}{\Leftrightarrow} gar_0(\sigma_0) \wedge gar_1(\sigma_1) \ ,$$

for all $\sigma \in \overrightarrow{IP}$, $\sigma_0 \in \overrightarrow{IP}_0$, $\sigma_1 \in \overrightarrow{IP}_1$ s.t. σ_0, σ_1 are valid projections of σ and the following holds

$$gar_0(\sigma_0) \Rightarrow ass_1(\sigma_1) \quad \text{and} \quad gar_1(\sigma_1) \Rightarrow ass_0(\sigma_0) \ . \qquad \square$$

Proposition 1. *The composition of composable MFIs is well-defined, i.e., it yields an MFI again, strongly time guarded, commutative, and associative.* □

Example 3. We compose the milling machine and transportation system MFIs from Examples 1 and 2 to $MFI_{mm} \otimes MFI'_{tr}$ given by $(\{en_1, en_2, en_{mm}\}, \{ex_1, ex_2, ex_{mm}\}, vol, val_{mm} \cup val_{tr}, novol, ass, gar)$. They are composable according to the above definition. According to Eq. (2) the volume function maps ex_1, en_{mm}, en_2, and ex_{mm} to an empty set. The volumes of en_1 and ex_2 do not change. Then, according to Eq. 3 $novol = vol_{mm}(en_{mm}) \cup vol_{mm}(ex_{mm})$. No assumptions are put on the environment: $ass = \text{true}$. The guarantee is given by gar'_{tr} and an additional constraint that en_1 accepts and ex_2 outputs at most one piece at a time every 75s: $gar(\sigma) = gar'_{tr}(\sigma) \wedge \forall k \in \mathbb{N} : |\sigma(en_1).(75 * k)| = |\sigma(ex_2).(75 * k + 64)| \leq 1$. □

Properties of Composition. An important feature of our framework is the preservation of plausibility properties from Sec. 3.2 upon composition. By this, it becomes sufficient to check for their satisfaction locally and only plausible systems can be built out of plausible components.

Proposition 2. *The following properties* remain preserved *in the composition $MFI = MFI_0 \otimes MFI_1$, provided MFI_0 and MFI_1 are composable: MFI is (1) sparse, (2) spatially plausible (produces streams of non-intersecting objects), and (3) movable (invariant resp. shape-preserving transformations).* □

5 Implementation Conformance

This section bridges between the abstract manufacturing systems (AMS) and the material flow specifications (MFI) by formalizing the notion of *implementation conformance*. More precisely, we define under which conditions an AMS is called a *realization* of an MFI. Informally, we interpret the MFI's entries as generators for material and require the AMS to generate corresponding output pattern at the exits.

For the remainder of this section let there be an AMS defined by inputs I_C and I_S, outputs O, interior N, sensor function S, and $F : \vec{I} \to \mathcal{P}(\vec{O} \times \mathcal{MF})$, as well as an MFI given by the tuple $(EN, EX, vol, val, novol, ass, gar)$. Let $\eta \in \overrightarrow{EN}$ such that $ass(\eta)$ holds, $i \in \vec{I}$, and $(o, f) \in F(i)$. Then the *spatio-temporal configuration* of the AMS is given by a sparse stream $stc(\eta, f) \in (\mathcal{MC} \cup \{\bot\})^{\mathbb{T}}$ defined as

$$stc(\eta, f).t \overset{\text{def}}{=} \begin{cases} \bot & \text{iff } \eta.t = \emptyset \wedge f.t = \text{id} \\ clear_N(f.t(prev) \cup \eta.t) & \text{otherwise} \end{cases} ,$$

where $\eta.t$ is shorthand for the set of material objects entering any of the entries at time t, $prev$ is the set of material from the last time where $stc(\eta, f)$ was not $\bot$ (defined as $\emptyset$ if no such time exists), and $clear_N$ defined for a material configuration $m \in \mathcal{MC}$ as

$$clear_N(m) \overset{\text{def}}{=} \{(v, s) \in m \mid v \cap N \neq \emptyset\} .$$

As both η and f are sparse (i.e., the number of non-trivial values until any time t is finite), the value of $prev$ and thus the spatio-temporal configuration is well-defined. The intuition is that the material is affected by the motion implied from the material flow f while new material is added in response to entry events from η. The $clear$ function discards material as soon as it leaves the scope of the AMS.

For the configuration $stc(\eta, f)$, the exit stream $\xi(stc(\eta, f)) \in (\mathcal{MC} \cup \bot)^{\mathbb{T}}$ captures the position of material leaving the interior of the AMS and is defined as

$$\xi(stc(\eta, f)).t \overset{\text{def}}{=} \begin{cases} \bot & \text{iff } stc(\eta, f) = \bot \\ (f.t(prev) \cup \eta.t) \setminus stc(\eta, f) & \text{otherwise} \end{cases} ,$$

where $prev$ is the same as before, i.e., refers to the last non-$\bot$ value of the stc function. Thus, ξ contains exactly those material objects discarded by $clear_N$ before and describes for each object the position and state when leaving the scope of the AMS. We call this exit stream *valid* for the MFI, if there is an exit event stream $\chi \in \overrightarrow{EX}$ such that $gar(\eta)(\chi)$ holds and for each $t \in \mathbb{T}$ the sets $\xi.t$ and $\bigcup_{x \in EX} \chi(x).t$ are equal.

Definition 4 (Implementation). *An AMS defined by I_C, I_S, O, N, S, and $F : \vec{I} \to \mathcal{P}(\vec{O} \times \mathcal{MF})$ is called an* implementation *of the MFI $(EN, EX, vol, val, novol, ass, gar)$, iff for all $\eta \in \overrightarrow{EN}$ with $ass(\eta)$, $i \in \vec{I}$, and $(o, f) \in F(i)$ with $\forall t \in \mathbb{T} : (i|_{I_S}).t = S(stc(\eta, f).t)$ the exit stream $\xi(stc(\eta, f))$ is valid for the MFI.* □

6 Application

We see the following application areas of our framework: (1) augmented simulation and run-time verification, (2) verification and, in particular, compositional verification,

(3) the correct-by-construction synthesis of AMS models conforming to a given MFI specification. While the latter two points belong to the future research directions, the prototypical implementation and evaluation of the simulation is presented here.

6.1 Augmented Simulation

Often, industrial-size systems with complex material flow cannot be simulated as a whole, due to the fact that a realistic physics simulation is a computationally complex task. A promising solution is to simulate selected fractions of the system, "subsystem under test", while abstracting the rest by corresponding MFI specifications. This would decrease the complexity since that MFIs reduce the material trajectories to the points of their expected observation. But still, the selected fractions would be simulated in a realistic environment, provided every abstracted subsystem is conform with its resp. MFI in sense of Sec. 5. This is because for our compositional semantics monotonicity resp. refinement can be shown analogous to results from [6,7,8].

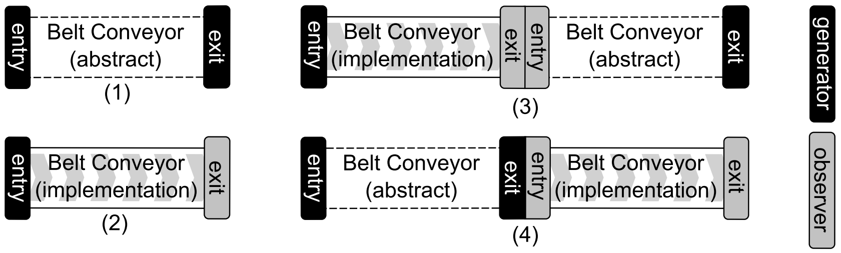


Fig. 2. Role of MFI Interface Points in Augmented Simulation

In the augmented simulation setup we obtain two types of components: *abstract* ones, which are specified by an MFI only, and *concrete* ones, which also have an implementation (AMS). Moreover, for validation by simulation we assign two different roles to interface points: *generators* and *observers*. A generator produces material flows in accordance with the resp. specification (ass or gar), while the observer checks the conformance to the specified flow. Fig. 2 depicts the possible configurations and the roles of the ports. Scenario (1) consists of only abstract components and thus all ports have to act as generators (there is no implementation which could affect the material flow). In the second scenario, where a concrete component is used, the entry still is a generator (the material has to be provided according to the precondition of the component), but the exit now is an observer which checks whether the resulting material stream matches the expectation. The interesting cases are (3) and (4) which explain the crossing between abstract and concrete components. In both cases the outer ports' roles are the same as in (1) and (2). In (3) material is passed from a concrete component to an abstract one, thus the inner ports are both observers, as material is provided by the concrete component. The opposite case (abstract to concrete) is depicted in (4). Here, the exit of the abstract component is used to generate a material stream and, thus, is a generator, while the entry of the concrete component still acts as an observer.

6.2 Tool Support and Case Study

We realized the concepts of MFIs and augmented simulation described here in the CASE tool AF/STEM.[3] AF/STEM is an extension of AutoFOCUS 3 which integrates automaton and data-flow based system descriptions with spatial models. AutoFOCUS allows the description of a system by data-flow diagrams consisting of components, typed ports (describing a component's interface), and channels, which connect ports. Components are either described using an automaton or by another data-flow diagram with further sub-components. By this, a component hierarchy is created.

AF/STEM extends this meta-model by *parts*, *detectors*, and *movers* (c.f., [1,2]), which are used to model the spatial properties of components and the reactions to collisions (e.g., between a transported object and the light-ray of a photo-electric barrier). Parts are spatial bodies, which describe the geometry of a component. Hybrid automata of AF/STEM components receive collision information between their detectors and any parts as a further input (e.g., an input variable representing a light ray is set to true when it collides with any other object). The position of parts can be changed by movers according to defined kinematic axes. These changes are also triggered by the automaton, which has dedicated mover variables as additional outputs. The changes of these variables are interpreted as direction and speed of movements in the spatial world (e.g., by a positive change of a mover variable every part colliding with the belt part of a conveyor is moved forward). We refer the reader to [2] for an in-depth description of AF/STEM. The meta-model of AF/STEM can be mapped to the formal AMS implementation model presented in Sec. 2.

To specify a piece of material, we use the same component-based description technique as for the system itself. By this, material incorporates both spatial configuration and a state, as described here. For a material component there can be multiple instances in the simulation, which have to be created and destroyed dynamically. This task is carried out by *MFI specifications* as presented here. Component have an MFI specification, which in turn consists of a number of interface points, an assumption, and a guarantee component. These elements directly correspond to the members of the MFI tuple from Sec. 3.1. Finally, an abstract component as described in Sec. 6.1 has no automaton, but only an MFI specification.

We used MFI specifications in a variety of academic and real-life models including separation and dislocating stations, quality gates, a tool changer of a milling machine, and the (un-)loading system of a grinding machine.

The concept of augmented simulation was evaluated on a model of an automotive assembly line. It consists of three conveyors, one for car bodies and two for wheels, and a pair of robots, which mount the wheels on the car. The workflow steps are: transport wheels, transport car body, mount wheels, transport car with mounted wheels. We modeled a pure abstraction, a pure implementation, and an abstraction with implemented robot and wheel conveyor. The behavior of the abstract model can be observed only when simulating in a step-wise manner. During continuous simulation the augmented model showed a lower resource usage in comparison with the pure implementation.

[3] `http://af3.in.tum.de/index.php/AF/STEM`

7 Related Work

This section summarizes work that is related to the formal model presented here.

Formal Models for Timed and Hybrid Systems. Depending on the level of abstraction chosen, manufacturing systems can be seen as special cases of timed or (if more control over flow conditions is required) hybrid systems. Thus, it seems reasonable to apply existing modeling techniques developed for these systems, such as timed [9] and hybrid automata [10]. Iversen et al. demonstrate in [11] how timed automata can be used to model a *Brick Sorter*, which consists of a conveyor belt, a color sensor and a kicking arm. Albeit this is an artificial example, with the "hardware" consisting of LEGO bricks, it demonstrates how timed automata can be applied to model manufacturing systems. This example also demonstrated the drawbacks of both timed and hybrid automata, as the bricks (material) have to be modeled as separate automata each, which encodes the interaction with the parts of the system. Thus, any extension of the system has to be reflected in the material automata, i.e., the model is not compositional with respect to the material flow abstraction.

Manufacturing Process Specifications. SADT [12] (Structured Analysis and Design Technique) is a modeling technique that allows the abstract specification of processes using box and arrow diagrams. The models can also be used to describe the material flow between several processing steps. Contrary to our approach, SADT has no formal semantics and does not include a notion of space. A more formal model is presented by Leuxner et al. in [13]. While the example chosen there is from the medical domain, the application to manufacturing processes in straightforward. This model, however, only supports a logical notion of material flow, while we support an explicit representation including spatial position and extension.

Formal Models for Mechatronic Systems. Many existing models for mechatronic systems lack a formal semantic foundation. A notable exception is Modelica [14], which describe systems in terms of (differential) equations combined with discrete state-based descriptions. Hierarchic composition of subsystems is supported via pins and connectors. The authors of this paper also describe a stream-based formalism for spatio-temporal mechatronic systems from the automation domain [1] and an extension which supports the modeling of hardware errors [15]. All of these models have in common that their focus is on the system itself and not the material flow. Material can only be expressed as another system part, leading to the same problems of composition as with timed/hybrid automata.

Material-Flow Simulations. In [16], Struss et al. describe a compositional mathematical model which allows the description of the material flow based on *inflow* and *outflow* rates of elements. The model only covers transportation (not modification) of material and can be used for material flow simulations using Matlab/Simulink. Compared to our approach, it does not describe the spatial positions of material, does not support assumption/guarantee reasoning on the level of individual material pieces, and does not support state changes in the material being processed.

8 Conclusion and Outlook

This paper described a model for manufacturing systems based on a material flow abstraction. The model is compositional and allows assumption/guarantee reasoning about the material flow. Furthermore, conformance between the material flow specification and an implementation model can be checked. We described a tool implementation which allows the simulation of material flow models in conjunction with more detailed abstract manufacturing models and reported on a first small case study.

Possible directions for future work include extending the model to also allow the modeling of *fractional goods*, such as liquids. Another goal is to improve tool support especially with respect to automatic conformance checking between abstract system descriptions and MFIs. A first step required for this is to detail an operationalized model which can be actually used for AMS/MFI description and find suitable decision procedures for them. First steps towards an operationalized model have already been taken in the context of a tool prototype's implementation.

References

1. Hummel, B.: A semantic model for computer-based spatio-temporal systems. In: Proc. of ECBS 2009 (2009)
2. Botaschanjan, J., Hummel, B., Lindworsky, A., Hensel, T.: Integrated behavior models for factory automation systems. In: Proc. of ETFA 2009 (2009)
3. Broy, M.: Refinement of time. Theoretical Computer Science 253, 3–26 (2001)
4. Müller, O., Scholz, P.: Functional specification of real-time and hybrid systems. In: Maler, O. (ed.) HART 1997. LNCS, vol. 1201, Springer, Heidelberg (1997)
5. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Proc. of ESEC/FSE 2009. ACM, New York (2001)
6. Abadi, M., Lamport, L.: Conjoining specifications. ACM Trans. Program. Lang. Syst. 17, 507–535 (1995)
7. Misra, J., Chandy, K.M.: Proofs of networks of processes. IEEE Trans. Softw. Eng. 7, 417–426 (1981)
8. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. 5, 596–619 (1983)
9. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126, 183–235 (1994)
10. Henzinger, T.: The theory of hybrid automata. In: Verification of Digital and Hybrid Systems. NATO ASI Series F: Computer and Systems Sciences, vol. 170, pp. 265–292. Springer, Heidelberg (2000)
11. Iversen, T.K., Kristoffersen, K.J., Larsen, K.G., Laursen, M., Madsen, R.G., Mortensen, S.K., Pettersson, P., Thomasen, C.B.: Model-checking real-time control programs: verifying Lego Mindstorms systems using UPPAAL. In: Proc. of ECRTS 2000 (2000)
12. Marca, D.A., McGowan, C.L.: SADT: Structured Analysis and Design Techniques. Mcgraw-Hill, New York (1987)
13. Leuxner, C., Sitou, W., Spanfelner, B., Thurner, V., Schneider, A.: Modeling work flows for building context-aware applications. Technical Report TUM-I0913, Technische Universität München (2009)
14. Tiller, M.: Introduction to Physical Modeling with Modelica. Springer, Heidelberg (2001)
15. Botaschanjan, J., Hummel, B.: Specifying the worst case - orthogonal modelling of hardware errors. In: Proc. of ISSTA 2009. ACM Press, New York (2009)
16. Struss, P., Kather, A., Schneider, D., Voigt, T.: A compositional mathematical model of machines transporting rigid objects. In: Proc. of ECAI 2008 (2008)

Specification and Verification of a MPI Implementation for a MP-SoC

Umberto Souza da Costa, Ivan Soares de Medeiros Júnior, and Marcel Vinicius Medeiros Oliveira

Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada
59.072-970, Natal, RN, Brazil
`{umberto,marcel}@dimap.ufrn.br`,
`ivsmjunior@gmail.com`

Abstract. System-on-Chip is a solution that integrates several components of a computer into a single chip substrate. Those systems are generally targeted for embedded applications and can increase their processing power by using multiple processors and an on-chip interconnection. STORM is a Multi-Processor System-on-Chip virtual platform which uses a basic implementation of the MPI standard to provide communication among their applications. STORM implements a small set of MPI routines for essential point-to-point and collective communication in order to provide more programmability and portability for the applications of the platform. In this work, we make use of CSP to build a formal model of those MPI routines and eliminate imprecision and ambiguities that may arise from their informal descriptions on the MPI standard. Also, we use the FDR model checker to ensure that the implemented routines have no errors introduced during the development process.

Keywords: CSP, concurrency, parallel computing, embedded systems.

1 Introduction

System-on-Chip (SoC) integrates several computer components into a single chip substrate. A SoC puts general purpose processors, digital signal processors (DSP), memory and I/O subsystems, and an internal communication subsystem together on the same chip. Those systems are usually targeted for embedded applications and can increase their processing power by using multiple processors and an on-chip interconnection to integrate them. So, a Multi-Core or Multi-Processor System-on-Chip (MP-SoC) [1] is a SoC where several general purpose processors execute in parallel on a same chip area [2].

STORM [3] is a MP-SoC virtual platform implemented in cycle accurate SystemC [4]. Processors of the platform communicate by using a Network-on-Chip (NoC) [5], a customizable interconnection option that presents good scalability [6]. The STORM configuration considered in this work uses a distributed memory with shared addressing space, implemented by means of several local memory blocks directly connected to processors. Processors on STORM communicate by using a MPI [7] implementation, intended as a mechanism to augment programmability and promote standardization of applications on the platform.

This work is concerned with the specification and verification of the MPI routines implemented on STORM, intended to improve the reliability of its applications. Two CSP [8] specifications are provided: one for the standard behavior of MPI routines and another for the routines implemented on STORM. The failures-divergences semantics [9] is the CSP model considered. Verifications are performed with the Failures-Divergences Refinement (FDR) [10] model checker.

Background is presented in Section 2, which provides information on CSP, on STORM and on the implemented MPI routines. Next, Section 3 presents the specifications produced and their verification with FDR. Section 4 compares our approach with related works. Finally, Section 5 presents final remarks.

2 Background

This section provides essential information on the specification language CSP and on the STORM platform, concerned with the inter-process communication.

2.1 The Specification Language CSP

A process algebra like CSP [8,9] can be used to describe systems composed of interacting components, which are independent self-contained processes with interfaces used to interact with the environment. Such formalisms provide a way to explicitly specify and reason about interaction between different components. Furthermore, phenomena that are exclusive to the concurrent world, that arise from the combination of components and not from the components alone, like deadlock and livelock, can be more easily understood and controlled using such formalisms. Tool support is another reason for the success of CSP in industrial applications, and consequently, for our choice to use it as the formal notation. For instance, FDR [10] provides an automatic analysis of correctness and of properties like deadlock and divergence.

The two basic CSP processes are `STOP` (deadlock) and `SKIP` (successful termination). The prefixing `c -> P` is initially able to perform only the event `c`; afterwards it behaves like process `P`. A Boolean guard may be associated with a process: given a predicate `g`, if the condition `g` is true, the process `g & c?x -> A` inputs a value through channel `c` and assigns it to the variable `x`, and then behaves like `A`, which has the variable `x` in scope; it deadlocks otherwise. It can also be defined as `if g then c?x -> A else STOP`. Multiple inputs and outputs are also possible. For instance, `c?x?y!z` inputs two values that are assigned to `x` and `y` and outputs the value `z`.

The sequence operator `P1;P2` combines processes `P1` and `P2` in sequence. The external choice `P1 [] P2` initially offers events of both processes. The performance of the first event resolves the choice in favor of the process that performsit. The environment has no control over the internal choice `P1 |~| P2`. The sharing parallel composition `P1 [| cs |] P2` synchronizes `P1` and `P2` on the channels in the set `cs`; events that are not listed occur independently. Processes composed in interleaving `P1 ||| P2` run independently. The event hiding operator `P \ cs` encapsulates the events that are in the channel set `cs`, which become no longer visible to the environment.

CSP provides finite iterated operators that can be used to generalize the binary operators of sequence, external and internal choice, parallel composition, and interleaving. Furthermore, we may instantiate a parameterized process by providing values for each of its parameters. For instance, we may have either `P(v)`, where `P = x:T @ Proc`, or, for reasoning purposes, we can write directly `(x:T @ Proc)(v)`. Apart from sequence, all the iterated operators are commutative and associative. For this reason, there is no concern about the order of the elements in the type of the indexing variable. However, for the sequence operator, we require this type to be a finite sequence. By way of illustration, the process `x:T @ P(x)` is the sequential composition of processes `P(v)`, with `v` taken from `T` in the order that they appear. Furthermore, `|~| x:T @ P(x)` is the internal choice of all process `P(v)` with `v` taken from `T` with no particular order. The semantics of the iterated interleaving `||| x:T @ P(x)` and parallel composition `[| cs |] x:T @ P(x)` are similar. For the latter, however, we also declare the synchronization channel set for all instances `P(v)`: they all synchronize in `cs`.

FDR also supports a functional language that provides us with the possibility of having some programming facilities within the specification. Among some built-in facilities of FDR's functional language we have set operations like set membership (`member`), set difference (`diff`), set `union`, set cardinality (`card`), sequence operators like `sizeof` that returns the sequence size, tuples, set and sequence comprehension, and pattern matching.

2.2 The STORM Platform

The STORM configuration considered on the distributed memory model is presented in Figure 1 (adapted from [11]). Processors have distinct cache modules for instructions (*ICache*) and data (*DCache*) in order to allow simultaneous access to those streams. The *Data Access Manager* (*DAMa*) provides a common interface between caches and other modules. All modules connected to *DAMa* are perceived as memory modules by the processor, each one associated with a specific address range. The memory module of a given processor cannot be accessed by other processors. Communication among processors is provided exclusively with the message exchange on the NoC through the *Communication Manager* (*CoMa*). The *CoMa* module sends and receives data by using the network. This module is able to provide the inter-processor communication with no software support, but that makes the development of applications more difficult. That is the reason why MPI routines have been implemented for STORM.

The *CoMa* has two buffers, one for sending (*OutBuffer*) and another for receiving (*InBuffer*) data through the NoC. The sizes of those buffers are configurable and can be different. The *CoMa* buffers are managed with the help of four registers, associated with memory addresses. Registers *Available SendBuffer*

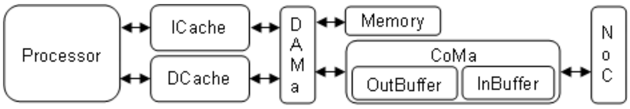


Fig. 1. STORM distributed memory model

and *Available RecBuffer* are used to inform the available space on the *OutBuffer* and *InBuffer*, respectively. Registers *SendBuffer Data* and *RecBuffer Data* are used to write and read data transmitted to and from the NoC.

In an inter-processor communication, the source processor first writes a 32-bit heading containing the NoC address of the destination processor and the length of the message into the *SendBuffer Data* register. After that, the processor also writes the data to be sent into the same register. Finally, the *CoMa* assembles the message and transmits it through the NoC. During the reception of the message, the *CoMa* discards the control information and stores the useful data into the *InBuffer* of the destination processor. The bounds of the messages stored in *InBuffer* must be controlled by software. The availability of the *SendBuffer Data* and *RecBuffer Data* registers also must be verified by the application software. Processors will get blocked if a writing operation is required on a full *SendBuffer Data* or if a reading operation is required on an empty *RecBuffer Data.*

The implementation of MPI routines in STORM counts on the hardware modules discussed so far. STORM has not an operating system at the current time, so that only static processes are permitted, one process per processor. Because the implemented MPI routines have no support from process and memory management software, such routines are loaded together with the end-user application and works as a software layer between it and the platform. Due to the current limitations of the STORM platform, only a subset of the MPI standard has been implemented at this time. However, the essential communication routines are available and make the development of parallel applications possible.

2.3 Implemented MPI Routines

The implementation of MPI routines in STORM assumes that each process can access only its local memory and all inter-process communication is done with message exchange. The implementation considers asynchronous versions of the send and receive routines, meaning that messages transmitted on the NoC are written and read to and from MPI buffers implemented in software. The transference of data between hardware and software buffers is implemented in software. The MPI buffers have fixed-sizes that are defined by the user of the platform. The user must also define the maximum size of the MPI messages and choose the root processor, which initiates and finalizes the MPI environment. The implementation provides the basic data types and constants of the MPI standard. Below, we present the MPI routines implemented on STORM:

- **MPI_Init()**: initializes the MPI environment. In accordance with the MPI standard, this routine must be called by each process before any other MPI routine. In the STORM implementation, this routine is called by every processor but only the root processor determines the ranks of processes.
- **MPI_Finalize()**: finalizes the MPI environment.
- **MPI_Comm_size()**: returns the number of processes in a given communicator. Only the default communicator *MPI_COMM_WORLD*, which contains all the processes on the MPI environment, is implemented.
- **MPI_Comm_rank()**: returns the rank of a process in a communicator.
- **MPI_Send()**: sends messages across the NoC. This routine copies the data from the MPI buffer to the hardware buffer (*SendBuffer Data*) of the processor that is the source of the communication. The message is sent through the NoC using hardware communication mechanisms provided by the platform.
- **MPI_Recv()**: receives a message from a processor on the NoC. This routine copies the data received by the hardware buffer (*RecBuffer Data*) of the processor that is the destination of the communication to the MPI buffer. The expected message must be selected from the messages in the MPI buffer.

STORM also implements routines for packing and unpacking structured data:

- **MPI_Pack()**: packs data into a contiguous memory storage.
- **MPI_Unpack()**: unpack data from a contiguous memory storage.
- **MPI_Pack_size()**: returns the number of bytes needed to pack a message.

More sophisticated routines are implemented for collective communications:

- **MPI_Broadcast()**: one-to-all communication routine.
- **MPI_Reduce()**: all-to-one communication routine.

The implementation of collective routines is based on the *recursive doubling* technique [12] to avoid bottlenecks on the root process and optimize the usage of the network. In *MPI_Broadcast()*, the rationale of this technique is to recursively split the range of destination processes $r = [1 \dots n]$ into ranges $r_1 = [1 \dots k-1]$ and $r_2 = [k \dots n]$ so that processes in r_1 and r_2 do not communicate with each other. In our implementation, instead of sending the message to every MPI process, the root process r sends the message only to processes with ranks $(r*2)$ and $(r*2+1)$. Those processes then execute the same procedure recursively, acting as roots for next level of communication, until each process receives the message. *MPI_Reduce()* is performed by reversing the direction and sequence of communications so that data from source processes are combined at intermediate processes before the final result is accumulated at the destination process.

3 Formalization of Routines

This section presents two specifications. The first one specifies the standard behavior of the MPI routines. The second specification describes the real behavior of the routines implemented for STORM. Together, those specifications allow us to check the implemented routines against the MPI standard with FDR.

3.1 Standard Routines

In accordance with the MPI Standard, each parallel process must be initialized with a call to *MPI_Init()*, prior calling any other MPI routine. The MPI standard does not prevent a call to any other routine to occur before initialization, but it establishes that such routines cannot be executed at that time and that the user must be notified with an error message in face of those calls. `MPINotInitialized` describes the behavior of the system before initialization:

```
MPI = MPINotInitialized
MPINotInitialized =
    (mpi_init -> (|~| r : MPIRanks @ mpi_rank.r -> mpi_init_ok
                  -> MPIInitialized(r)))
  [](mpi_finalize -> mpi_finalize_error -> MPINotInitialized)
  [] ... [](mpi_reduce?sendbf?recvbf?ct?dt?op?root?comm ->
     mpi_reduce_error -> MPINotInitialized)
```

`MPINotInitialized` corresponds to the MPI user, the external agent that chooses the routine to be executed. The event *mpi_init* describes a call to *MPI_Init()* and takes the system to the state described by `MPIInitialized(r)`, where *r* stand for the rank of the MPI process. During initialization, the system randomly chooses a rank `r` from the set of valid MPI ranks (`MPIRanks`) and communicates it by using the event `mpi_rank`. After that, the system uses the event `mpi_init_ok` to inform that the environment is initialized and behaves as `MPIInitiated(r)`. Note that events that correspond to a call to any other routine in `MPINotInitialized` produce error events and take the system back to the pre-initialization state. `MPIInitialized` represents the MPI environment after its proper initialization, when the MPI routines execute in interleaving:

```
MPIInitialized(r) =
 (MPICommSize ||| MPICommRank(r) ||| MPISend(r) ||| MPIRecv(r) |||
  MPIPack ||| MPIUnpack ||| MPIPackSize |||
  MPIBcast(r) ||| MPIReduce(r)) [|MPIEvents|] (MPIControl)
```

Each routine is specified as a CSP process. Those processes are synchronized on `MPIEvents`, a set of events for describing calls, successful and unsuccessful terminations of routines. Obviously `MPIInitialized` could be merged into `MPIControl`, but we use a separate process per routine for the sake of a modular and clearer specification. `MPIControl` specifies the external selection of routines:

```
MPIControl =
   (mpi_init -> mpi_init_error -> MPIControl)
 [](mpi_finalize -> SKIP)
 [] ... [](mpi_reduce?sendbf?recvbf?ct?dt?op?root?comm ->
        (  (mpi_reduce_ok?recvbf -> MPIControl)
         [](mpi_reduce_error -> MPIControl)))
```

Except for `mpi_init` and `mpi_finalize`, `MPIControl` produces either an event to confirm the successful termination of the routine or an event indicating that it has failed. The event `mpi_init` always produces an error event because the system is already initialized in that process. On the other hand, `mpi_finalize` produces a `SKIP` to represent the termination of the MPI environment.

MPI_Init() and MPI_Finalize(). The event `mpi_init` represents a call to *MPI_Init()*. Different from other routines, *MPI_Init()* is not specified in a separate CSP process, but in `MPINotInitialized`. The latter process assigns a rank to the process and becomes `MPIInitialized`. Analogously, *MPI_Finalize()* is specified in `MPIControl`, using the event `mpi_finalize` to produce a `SKIP`.

MPI_Comm_size() and MPI_Comm_rank(). Those routines have similar specifications. `MPICommSize` specifies *MPI_Comm_size()*. After validating its parameters, this process either informs the size `s` of the communicator on `mpi_comm_size_ok` or produce the error event `mpi_comm_size_error`.

```
MPICommSize =
   mpi_comm_size?comm?size ->
      if member(comm, MPICommunicators) and member(size, MPIValidAdd)
      then (|~| s : MPICommSizes @ mpi_comm_size_ok.s -> MPICommSize)
      else (mpi_comm_size_error -> MPICommSize))
```

The specification of the communicator's size is non-deterministic since it depends on the parallel application. The types `MPICommunicators, MPIValidAdd` and `MPICommSizes` are defined by the MPI implementation to represent communicators, non-null memory addresses and communicator sizes, respectively. The specification of *MPI_Comm_rank()* is in `MPICommRank(r)`, where `r` is the rank of the process:

```
MPICommRank(r) =
   mpi_comm_rank?comm?rank ->
     (if member(comm, MPICommunicators) and member(rank, MPIValidAdd)
      then (mpi_comm_rank_ok.r -> MPICommRank(r))
      else (mpi_comm_rank_error -> MPICommRank(r)))
```

`MPICommRank` first verifies if its parameters are valid. Next, it uses either the event `mpi_comm_rank_ok` to inform the rank `r` of the process or produces the event `mpi_comm_rank_error` to indicate failure on the execution of the routine.

MPI_Send() and MPI_Recv(). `MPISend` specifies *MPI_Send()* and considers both synchronous and asynchronous behaviors. `MPISend` verifies the parameters of the routine and chooses between the synchronous (`MPISyncSend`) and the asynchronous (`MPIAsyncSend`) behaviors. That decision is non-deterministic since it depends on the MPI implementation.

In `MPISyncSend` the communication depends on the agreement between source and destination processes. Event `sync` notifies the destination `dest` that source `r` is waiting for sending a message with tag `tag`, where `dest` and `r` are ranks. After synchronization, event `msg` is used to represent the sending of the message in fact. This event contains the tag, the communicator, the size of the message (defined as `ct * dtsize(dt)`) and the message itself, stored in buffer `bf`. Finally, the process `MPISyncSend` is notified with a successful (`msg_ok`) or unsuccessful (`msg_error`) termination.

```
MPISyncSend(r, bf, ct, dt, dest, tag, comm)=
   sync.r.dest.tag -> msg!(tag, comm, (ct*dtsize(dt)), bf) ->
      (msg_ok -> mpi_send_ok -> MPISend(r)
      [] msg_error -> mpi_send_error -> MPISend(r))
```

In `MPIAsyncSend` the source assumes the communication to be accomplished after storing the message into the MPI buffer. After that, the source is free to continue executing and the message can be read by the destination later:

```
MPIAsyncSend(r, bf, ct, dt, dest, tag, comm) =
  add.((dest, (ct*dtsize(dt))), r, (ct, dt, tag, comm, bf))->
     mpi_send_ok -> MPISend(r)
```

The bufferization of the message is represented as event `add`. The source will get blocked in the case the buffer is not available for writing. After storing the message, `MPIAsyncSend` receives the event `mpi_send_ok` and executes `MPISend`.

`MPIReceive` specifies `MPI_Recv()`. It receives several parameters on event `mpi_receive`, including the rank of the source and the message tag. These parameters can be used to define a specific source and a specific tag for the expected message, or can be set as `ANY_SOURCE` and `ANY_TAG` to indicate that a message with any source and any tag matches the receive. `MPIReceive` first verifies parameters and then executes either the synchronous (`MPISyncRecv`) or the asynchronous (`MPIAsyncRecv`) receive operation.

In `MPISyncRecv` event `sync` contains the source `src`, the destination `r` and the tag `tag` of the message. If `src` is `ANY_SOURCE`, the process chooses a source `s` by using the external choice operator. A similar procedure is used to define the tag `t` of a message with `ANY_TAG`. Finally, event `msg` is used to receive the selected message. At this point, if the size of the message is correct the process produces an event to inform the source that the message has been received (`msg_ok`), extracts the buffer of the message (`mpi_recv_ok`) and turns to its initial state. Error events are executed when something goes wrong during those verifications.

```
MPISyncRecv(r, bf, ct, dt, src, tag, comm) =
  (if src == ANY_SOURCE then
    ([] s : diff(MPIRanks, {r}) @
     if tag == ANY_TAG then ([] t : MPITags @ sync.s.r.t -> SKIP)
     else (sync.s.r.tag -> SKIP))
   else (if tag == ANY_TAG then ([] t : MPITags @ sync.src.r.t -> SKIP)
         else (sync.src.r.tag -> SKIP)));
  (msg?m -> (if message_size(m) > (ct*dtsize(dt)) then
                msg_error -> mpi_recv_error -> MPIRecv(r)
             else msg_ok -> mpi_recv_ok!extract_buffer(m) -> MPIRecv(r)))
```

`MPIAsyncReceive` specifies the buffer-based receive. Event `rem` removes the expected message `m` from the MPI buffer. The message `m` is the one with destination `r` and tag `tag`, chosen deterministically by `filter_msg`. Event `mpi_recv_ok` extracts the message and then `MPIAsyncReceive` becomes `MPIRecv`.

```
MPIAsyncReceive(r, bf, ct, dt, src, tag, comm) =
  [] m : filter_msg(PackBuffer, src, tag, r, comm) @ rem.m ->
       (if (ct*dtsize(dt)) < message_size(m) then
          add.m -> mpi_recv_error -> MPIRecv(r)
        else mpi_recv_ok!extract_buffer(m) -> MPIRecv(r))
  [] (card(filter_msg(PackBuffer, src, tag, r, comm)) == 0) &
    (mpi_recv_error -> MPIRecv(r))
```

This process checks if the size of the message is correct, as done in MPISync-Recv. In addition, if the size of the extracted message is erroneous the message is taken back to the buffer. `PackBuffer` stands for all messages in the system.

MPI_Pack(), MPI_Unpack() and MPI_PackSize(). *MPI_Pack()* and *MPI_Unpack()* are specified in `MPIPack` and `MPIUnpack`. Such processes first check parameters. If parameters are correct, they call events that represent the packing (`pack.inbf.outbf`) or unpacking (`unpack.inbf.outbf`) of messages, followed by events `mpi_pack_ok` or `mpi_unpack_ok`, respectively. Otherwise they inform failures: `mpi_pack_error` or `mpi_unpack_error`, respectively.

MPI_PackSize() is specified in `MPIPackSize`. After checking parameters, it returns event `pack_size!(ct * dtsize(dt))`, where `ct` and `dt` stand for the number and datatype of items in the buffer. Event `mpi_pack_size_error` occurs if parameters are incorrect or the size of the package is invalid.

MPI_Bcast() and MPI_Reduce(). *MPI_Bcast()* and *MPI_Reduce()* are specified in `MPIBcast` and `MPIReduce`. In the first routine the root process broadcasts a message to all process in a communicator. The MPI standard does not establish how the communication takes place, but that each destination process must have a copy of the message at termination. In order to allow different implementations, the standard specification of broadcast is based on the following idea: processes with a copy of the broadcast message are in set `S`, initialized as `{r}` where `r` stands for the rank of the broadcast root; processes without a copy of the broadcast message are in set `D`, initialized as `diff(MPIRanks, {r})`; MPIBcast non-deterministically chooses processes `s: S` and `d: D`, uses `MPISend` to send the message from `s` to `d`, and then update `S` and `D` to `union (S, {d})` and `diff(D, {d})`, respectively; `MPIBcast` applies that procedure until `D` is empty.

The specification of *MPI_Reduce()* is similar: processes with elements yet not combined are in set `S`, initialized as `diff(MPIRanks,{r})`; MPIReduce non-deterministically chooses processes `s,d: S`, uses `MPISend` to send the element from `s` to `d`, combines the elements, stores the combined value into the input buffer of `d` and then updates `S` to `diff(S,{s})`; `MPIReduce` applies that procedure until `card(S)==1`, when `MPIReduce` sends the element from `u:S` to `r`, which combines elements and stores the combined value in the output buffer of `r`.

3.2 Implemented Routines

The specifications shown in this section have been produced from the STORM implementation and are going to be checked against the MPI standard. `I_MPI` gives the specification of each MPI process implemented for STORM. Every MPI process is the interleaving of CSP process for the MPI routines:

```
I_MPI = ((I_MPIInit [|InitEvents|]
          ((I_MPICommSize ||| I_MPICommRank ||| I_MPISend |||
            I_MPIRecv ||| I_MPIPack ||| I_MPIUnpack ||| I_MPIPackSize)
            [|CollectiveOpEvents|] (I_MPIBcast ||| I_MPIReduce)))
         \ HiddenEvents)[|FinalizeEvents|] I_MPIFinalize
```

`I_MPIInit` uses events `InitEvents` to inform about ranks and initialization. `I_MPIBcast` and `I_MPIReduce` uses events `CollectiveOpEvents` to synchronize themselves with `I_MPISend` and `I_MPIRecv` during collective communications. On the other hand, `I_MPIFinalize` uses events `FinalizeEvents` to specify that the execution of all the other processes cannot be interleaved. Finally, `HiddenEvents` are events hidden from external agents because they represent internal communications, such as informing about ranks and the MPI environment, and are also used to integrate individual and collective communications.

MPI_Init() and MPI_Finalize(). `I_MPIInit` initializes the MPI environment, sets the rank and notice the initialization. Other routines use `rank` and `initiated` to know the rank and the state of the environment, respectively.

```
I_MPIInit = mpi_init -> mpi_rank?r -> mpi_init_ok -> On(r)
            [] initiated!false -> I_MPIInit
On(r) = initiated!true -> On(r)
        [] rank!r -> On(r) [] mpi_init -> mpi_init_error -> On(r)
```

In the opposite, `I_MPIFinalize` finalizes the MPI environment and controls the non-interleaved execution of processes for the other MPI routines:

```
I_MPIFinalize =
   mpi_init -> ( mpi_init_ok -> I_MPIFinalize
                [] mpi_init_error -> I_MPIFinalize)
   [] mpi_comm_size?comm?size -> (mpi_comm_size_ok?s -> I_MPIFinalize
                                  [] mpi_comm_size_error ->
                                       I_MPIFinalize)
   []...[] mpi_finalize -> SKIP
```

Each routine has a specification process that synchronises with `I_MPIFinalize`.

MPI_Comm_Size() and MPI_Comm_Rank(). Parameterized routines use checking processes to consult the state of the MPI environment and to check parameters (named as `Check_Comm_Size`, `Check_Comm_Rank`, `Check_Send` and so on). Those checking processes either return `true` to indicate that execution conditions are satisfied or `false` otherwise. Next, we present `I_MPICommSize` to illustrate the general structure used in the specification:

```
I_MPICommSize =
  ((Check_Comm_Size [|Comm_SizeEvents|] Comm_Size) \ {|check_comm_size|})
Comm_Size = mpi_comm_size?comm?size -> check_comm_size?x ->
   (if x then mpi_comm_size!MPI_COMM_WORLD_SIZE -> Comm_Size
    else mpi_comm_size_error -> Comm_Size)
```

Comm_Size synchronizes with Check_Comm_Size to make sure that the execution conditions are satisfied and then returns the size of the communicator. Remember that only the communicator MPI_COMM_WORLD is supported in STORM. The specification of *MPI_ Comm_ Rank()* is similar.

MPI_Send() and MPI_Recv(). I_MPISend specifies the asynchronous send routine implemented in STORM. It can be called by the user or by collective communication routines. Event mpi_send and mpi_csend describes user calls and calls from collective communication routines, respectively.

```
I_MPISend = (((Send [|SendEvents|] Check_Send)
               [|PackEvents|] PackHeading) \HiddenSendEvents)
Send = mpi_send?bf?ct?dt?dest?tag?comm ->
         SendOp(bf, ct, dt, dest, tag, comm, PP)
       [] mpi_csend?bf?ct?dt?dest?tag?comm ->
         SendOp(bf, ct, dt, dest, tag, comm, CL)
SendOp(bf, ct, dt, dest, tag, comm, k) = check_send?x ->
  (if x then
      pack_msg!bf.ct.dt.dest.tag.comm ->
      pack_ok?m -> SendPks(m, ct, dt, dest, tag, comm);
     (k == PP & mpi_send_ok -> Send [] k == CL & mpi_csend_ok -> Send)
   else (k == PP & mpi_send_error -> Send
         [] k == CL & mpi_csend_error -> Send))
```

SendOp sends the message in fact. It uses PackHeading to prepare the message (pack_msg) and receive the message produced (pack_ok). Next, SendPks splits the message into packages, transmits them across the NoC and inserts the message into the buffer of the destination.

I_MPIRecv specifies *MPI_ Recv()*. It executes Recv to distinguishes individual (mpi_recv) and collective receives (mpi_crecv) and then executes RecvOp.

```
I_MPIRecv = ((Recv [|RecvEvents|] Check_Recv)\ HiddenRecvEvents)
Recv = mpi_recv?bf?ct?dt?src?tag?comm -> RecvOp (bf,ct,dt,src,tag,PP)
   [] mpi_crecv?bf?ct?dt?src?tag?comm -> RecvOp (bf,ct,dt,src,tag,CL)
RecvOp (bf,ct,dt,src,tag,k) = check_recv?x ->
   (if x then RecvPks(ct,dt); ReadBuf(src, tag,(ct* dtsize(dt)), k);Recv
    else mpi_recv_error -> Recv)
```

RecvOp performs verifications, receives the packages transmitted on the NoC (RecvPks) and reads the message from the MPI buffer (ReadBuf). ReadBuf uses the rank to reads the appropriate message. The received data is made available with mpi_recv_ok (individual receive) and mpi_crecv_ok (collective receive).

MPI_Pack(), MPI_Unpack() and MPI_PackSize. Processes `I_MPIPack`, `I_MPIUnpack` and `I_MPIPackSize` specify the implementation of *MPI_ Pack()*, *MPI_ Unpack()* and *MPI_ PackSize()*. Those processes are similar to the ones for the standard behaviors of those routines and therefore are not presented here.

MPI_BCast() and MPI_Reduce(). `I_MPIBCast` and `I_MPIReduce` specify the collective communication routines. Their implementations are inspired on the *recursive doubling* technique and rely on `I_MPISend` and `I_MPIRecv`.

```
I_MPIBcast = (((BCast [|BCastEvents|] Check_BCast)
               [|BCastBufEvents|] BCastBuf) \ HiddenBCastEvents)
BCast = mpi_bcast?bf?ct?dt?root?comm -> check_bcast?x ->
       (if x then rank?r -> (BCastRecv (bf, ct, dt, root, comm, r);
         (access_buf?nb -> (Dest1 (nb, ct, dt, root, comm, r);
                           Dest2 (nb, ct, dt, root, comm, r))
          [] root_process -> (Dest1 (bf, ct, dt, root, comm, r);
                              Dest2 (bf, ct, dt, root, comm, r)));
         (mpi_bcast_ok -> BCast))
        else mpi_bcast_error -> BCast)
```

`BCast` verifies parameters and retrieves the rank `r`. Next, it assumes `r` is not the root and tries to receive the message from root, in `BCastRecv`. `BCast` then synchronizes with `BCastBuf` to choose either `access_buf` or `root_process`, in order to determine the buffer with the contents to be broadcast (`nb` or `bf`). Non-root process uses `access_buf` to access the buffer `nb`. The root process is indicated by `root_process` and broadcasts the contents of `bf`. Finally, `Dest1` and `Dest2` uses `I_MPISend` to send the contents of the chosen buffer to processes with ranks (`r * 2`) and (`r*2+1`), respectively. Successful and unsuccessful terminations are indicated by events `mpi_bcast_ok` and `mpi_bcast_error`, respectively.

```
I_MPIReduce = ((((Reduce [|ReduceOpEvents|] ReduceOp)
                [|EvtsReduce|] Check_Reduce)
               [|ReduceBufEvents|] ReduceBuf) \HiddenReduceEvents)
Reduce = mpi_reduce?sb?rb?ct?dt?op?root?comm -> check_reduce?x ->
   (if x then rank?r -> operand!sb -> (RecvOp1(rb,ct,dt,root,comm,r);
      (access_buf?nb1 -> operator!(nb1,op) ->
           RecvOp2 (nb1, ct, dt, root, comm, r)
       [] no_src -> RecvOp2 (rb, ct, dt, root, comm, r));
      ((access_buf?nb2 -> operator!(nb2, op) -> SKIP [] no_src -> SKIP);
      (result?res -> ReduceSend (res, ct, dt, root, comm, r))))
    else mpi_reduce_error -> Reduce)
```

`Reduce` first verifies parameters, access the rank `r` of the process and uses the contents of buffer `sb` as the first operand of the reduction (event `operand`, synchronized with `ReduceOp`). Next, it tries to receive the operand sent by the first source in `RecvOp1`. `Reduce` then synchronizes with `ReduceBuf` on `access_buf` to determine if the first source could be found. If so, `Reduce` applies the operator (event `operator`, synchronized with `ReduceOp`). Otherwise, `Reduce` tries to read the operand sent by the second source (`RecvOp2`) and operate over it. If there is no a second source, `Reduce` just skips. Finally, the reduction result is read on `result` (synchronized with `ReduceOP`) and sent to the root by `ReduceSend`.

3.3 Verification

In this work, we used FDR to check the MPI implementation against the MPI standard. Furthermore, we also checked if the implementation is deterministic, deadlock free and livelock free. Verifications were performed with FDR version 2.82 on a 1.67 GHz Intel Core 2 Duo with 2GB of RAM.

The implementation `I_MPI` was proved to be deterministic. This is due to the fact that the test was done with a single instance of the library. For this reason, the value of `rank` is always the same. Otherwise, if we run the check with more than one instance of the library we get a non-deterministic system because the value of the rank for different execution are non-deterministically chosen. This, however, is not an incorrect behavior because different rank values do not affect the behavior of the other routines. The next verification showed us that the implementation `I_MPI` is not deadlock free. This was due to the fact that the standard specifies that when a process concludes the MPI environment, no other event may be executed. So, if the event `mpi_finalize` happens, no other event is accepted characterizing a deadlock. After that, the implementation `I_MPI` was proved to be livelock free. Finally, we checked that the implementation `I_MPI` behaves accordingly (refines) the specification of the standard. All these checks were achieved in a relatively quick interval as shown in Table 1, where we present the data of the analysis like time spent in each verification and the number of states investigated by FDR to yield the answer.

Table 1. FDR Analysis Data

Assertion	Time (s)	Number of States
assert I_MPI :[deterministic [FD]]	110	6914
assert I_MPI :[deadlock free [FD]]	110	842
assert I_MPI :[divergence free]	108	6914
assert MPI [FD= I_MPI	150	6914

As expected, the time spent in the verification of properties like non-determinism, deadlock freedom and livelock freedom were almost the same since the states explored were those of the implementation only. Nevertheless, the refinement check took longer because both the refinement and the implementation states are explored and compared. The table also shows that the implementation state space contains 6914 possibilities, which would have to be manually explored if it were not for an automatic model checking as we did here.

4 Related Work

The formalization of MPI is found in many works. In [13], the authors present the formal specification of 42 MPI routines in TLA+ [14], clarifying the points omitted in the informal MPI description. The specification of that set of routines is augmented in [15]. The work presented in [16,17,18] is focused on modeling and verifying MPI programs instead of MPI routines themselves. Such approach is mainly interested in detecting deadlocks. In [19], Carter and Gardner present a framework (CSP4MPI) for message-passing high-performance computing (HPC) programming, which is designed to hide the complexity of parallel programming for HPC. Their work focused on the development of HPC programs instead of guaranteeing the correctness of a particular MPI implementation, as done in this paper. Carter and Gardner provide a CSP abstraction layer on the top of the MPI primitives. For that, they developed a C++ library, focused on LAM/MPI [20], that provides a CSP-based process model and a set of candidate solutions for HPC programmers not familiarized with CSP. They also proposed a selective formalism approach in which the CSP-specified code plays the role of a control backbone that invokes user defined functions that calculate the desired results.

In [21], authors model and verify correctness of parallel algorithms of the Multi Purpose Daemon (MPD), a parallel process management system consisting of processes connected by lower-level UNIX operations. MPD is not MPI-specific, although it is part of some MPI implementations. The work concentrates on algorithms such as those for daemon ring establishment and barrier mechanisms. PROMELA is used as the modeling language and SPIN is used for simulation and verification. On the other hand, our work is concerned with MPI routines themselves and considers implementation on a MP-SoC platform.

5 Final Remarks

In our work the communication routines were implemented in SystemC and specified in CSP. Due to the difficulty in representing data in CSP, some routines were not specified in detail. For example, the specification of routines for packing and unpacking messages does not describe that the packed and unpacked data must be the same. Two CSP specifications were presented in this work: one for the standard behavior of MPI routines and another for the routines implemented on STORM. Those specifications were produced in a non-automated manner, by analyzing the informal description of the MPI routines and their implementation on STORM. So, the proved properties are guaranteed for the implementation specification, not for the real implementation. FDR was used to verify those specifications. FDR compared the standard specification with the implementation specification, with the aim of revealing implementation-specific properties and casually properties not in conformance with the MPI standard. FDR showed that the implementation meets the properties of the standard and is a valid refinement of the standard specification. Also, it proved that the implementation is deterministic and livelock free.

Our work can be improved with the verification of a larger number of properties of the MPI implementation, the specification of more MPI routines and also by representing the data handled in MPI routines. In addition, the usage of tools to automatically translate the source-code of routines into a CSP specification can also increase the accuracy of this work. The tool proposed in [13] for extracting models from C programs endowed with MPI routines gives directions to the creation of a similar tool for systems written in SystemC.

References

1. Loghi, M., Angiolini, F., Bertozzi, D., Benini, L., Zafalon, R.: Analyzing On-Chip Communication in a MPSoC Environment. In: Conference on Design, Automation and Test in Europe, pp. 752–757. IEEE Computer Society, Washington (2004)
2. Jerraya, A.A., Wolf, W.: The What, Why, and How of MPSoCs. In: Jerraya, A.A., Wolf, W. (eds.) Multiprocessor Systems-on-Chips, pp. 1–18. Morgan Kauffman, San Francisco (2005)
3. Girão, G., Oliveira, B.C., Soares, R., Silva, I.S.: Cache Coherency Communication Cost In A NoC-Based Mp-Soc Platform. In: 20th annual conference on Integrated Circuits and Systems Design, pp. 288–293. ACM, New York (2007)
4. IEEE Comp. Society: IEEE Standard SystemC Language Reference Manual (2006)
5. Benini, L., De Micheli, G.: Networks on Chips: A New SoC Paradigm. Computer 35, 70–78 (2002)
6. Zeferino, C.A., Kreutz, M.E., Carro, L., Susin, A.A.: A Study on Communication Issues for Systems-on-Chip. In: 15th Symposium on Integrated Circuits and Systems Design, pp. 121–126. IEEE Computer Society, Washington (2002)
7. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI: The Complete Reference. The MIT Press, Massachusetts (1998)
8. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Upper Saddle River (1985)
9. Roscoe, A.W., Hoare, C.A.R., Bird, R.: The Theory and Practice of Concurrency. Prentice Hall, Upper Saddle River (1997)
10. Formal Systems Ltd: Failures-Divergence Refinement. FDR2 User Manual (2005)
11. Oliveira, B.C.: Simulação de reservatórios de petróleo em ambiente MPSoC. Master's Dissertation, Pós-Graduação em Sistemas e Computação, UFRN (2009)
12. Van de Velde, E.F.: Concurrent Scientific Computing. Springer, New York (1994)
13. Palmer, R., Delisi, M., Gopalakrishnan, G., Kirby, R.M.: An Approach to Formalization and Analysis of Message Passing Libraries. In: Leue, S., Merino, P. (eds.) FMICS 2007. LNCS, vol. 4916, pp. 164–181. Springer, Heidelberg (2008)
14. Lamport, L.: Specifying Concurrent Systems with TLA+. In: Broy, M., Steinbrüggen, R. (eds.) Calculational System Design. NATO Science Series F, vol. 173, pp. 183–247. IOS Press, Amsterdam (1999)
15. Li, G., Delisi, M., Gopalakrishnan, G., Kirby, R.M.: Formal specification of the MPI-2.0 standard in TLA+. In: 13th Symposium on Principles and Practice of Parallel Programming, pp. 283–284. ACM Press, New York (2008)
16. Siegel, S.F., Avrunin, G.S.: Modeling MPI programs for verification. Technical report, Department of Computer Science, University of Massachusetts (2004)
17. Siegel, S.F., Avrunin, G.S.: Verification of MPI-Based Software for Scientific Computation. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 286–303. Springer, Heidelberg (2004)

18. Siegel, S.F.: Model Checking Nonblocking MPI Programs. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 44–58. Springer, Heidelberg (2007)
19. Carter, J.D., Gardner, W.B.: A Formal CSP Framework for Message-Passing HPC Programming. In: Canadian Conference on Electrical and Computer Engineering (CCECE), pp. 1466–1470. IEEE Computer Society, Washington (2006)
20. Burns, G., Daoud, R., Vaigl, J.: LAM: An Open Cluster Environment for MPI. In: Supercomputing Symposium, pp. 379–386. ACM Press, New York (1994)
21. Matlin, O.S., Lusk, E.L., McCune, W.: SPINning Parallel Systems Software. In: Bošnački, D., Leue, S. (eds.) SPIN 2002. LNCS, vol. 2318, pp. 213–220. Springer, Heidelberg (2002)

Testing of Abstract Components

Bilal Kanso[1,2], Marc Aiguier[1], Frédéric Boulanger[2], and Assia Touil[2]

[1] École Centrale Paris
Laboratoire de Mathématiques Appliquées aux Systèmes (MAS)
Grande Voie des Vignes F-92295 Châtenay-Malabry
{marc.aiguier,bilal.kanso}@ecp.fr
[2] SUPELEC Systems Sciences (E3S) - Computer Science Department
3 rue Joliot-Curie F-91192 Gif-sur-Yvette cedex
{frederic.boulanger,assia.touil}@supelec.fr

Abstract. In this paper, we present a conformance testing theory for Barbosa's abstract components. We do so by defining a trace model for components from causal transfer functions which operate on data flows at discrete instants. This allows us to define a test selection strategy based on test purposes which are defined as subtrees of the execution tree built from the component traces. Moreover, we show in this paper that Barbosa's definition of components is abstract enough to subsume a large family of state-based formalisms such as Mealy machines, Labeled Transition Systems and Input/Output Labeled Transition Systems. Hence, the conformance theory presented here is a generalization of the standard theories defined for different state-based formalisms and is a key step toward a theory of the test of heterogeneous systems.

Keywords: Component based system, Coalgebra, Monad, Trace semantics, Transfert function, Conformance testing, Test purpose.

1 Introduction

The design of complex software systems relies on the hierarchical composition of subsystems which may be modeled using different formalisms [12]. These subsystems can be considered as state-based components whose behavior can be observed at their interface. In order to model such components in an abstract way, we use a definition introduced by Barbosa in [1,19]. The interest of this definition is twofold: first, it can be used to describe differents kinds of state-based formalisms, and second, it extends Mealy machines, which, following Rutten's work [11], allows us to define a trace model over components using causal transfer functions. Barbosa defines a component as a coalgebra over the endofunctor $\mathcal{H} = T(Out \times \_)^{In}$ where T is a monad[1], and In and Out are two sets of elements which denote respectively inputs and outputs of the component. Hence, Barbosa's definition of a component is an extension of Mealy automata [18], which are efficient to specify the behavior of components deterministically. The

[1] The definitions and notations used in this paper are recalled in Section 2.

role of the T monad is to take into account in a generic way various computation structures such as non-determinism, partiality, etc [21]. Therefore, Barbosa's definition of a component allows us to model components independently of any computation structure but also independently of the state-based formalisms classically used to specify software components. Indeed we show in this paper that state-based formalisms such as Mealy automata, Labeled Transition Systems and Input-Output Symbolic Transitions [8,9] can be embedded into Barbosa's definition by a suitable choice for the monad T. Moreover, this way of modeling the behavior of components allows us, following Rutten's works [11], to define a trace model over components by causal transfer functions. Such functions are dataflow transformations of the form: $y = \mathcal{F}(x, q, t)$ where x, y and q are respectively the input, output and state of the component under consideration, and t stands for the time which is considered here as discrete.

Indeed, defining a trace model from causal functions allows us, first to show the existence of a final coalgebra in the category of coalgebras over a signature $T(Out \times \_)^{In}$ under some sufficient conditions on the monad T, and second, to define a conformance testing theory for components, which is the main contribution of this paper. Final coalgebras are important because their existence is the key of *co-induction*, a powerful reasoning principle. Following some previous works by some authors of this paper [9], we define test purposes as particular subtrees of the execution tree built from our trace model for components. Then, we define an algorithm for generating test cases from a test purpose. Like in [9], this algorithm is given as a set of inference rules. Each rule is dedicated to the handling of an observation of the system under test (SUT) or of a stimulation sent by the test case to the SUT.

The paper is structured as follows: Section 2 recalls the basic notions of the categorical theory of coalgebras and monads that are used in this paper. Then, Section 3 recalls Barbosa's definition of components and introduces our trace model from causal transfer functions. The formalization of components as coalgebras allows us to extend standard results connected to the definition of a terminal component. Section 4 presents our conformance testing theory for components, and Section 5 gives the inference rules for generating test cases.

2 Preliminaries

This paper relies on many terms and notations from the categorical theory of coalgebras and monads. We briefly introduce them here, but interested readers may refer to textbooks such as [2,7,17].

2.1 Categories, Functors and Natural Transformations

A **category** $\mathbb{C}$ is a mathematical structure consisting of a collection of objects $\mathsf{Obj}(\mathbb{C})$ and a collection of maps or morphisms $\mathsf{Hom}(\mathbb{C})$. Each map $f : X \to Y$ has a domain $X \in \mathsf{Obj}(\mathbb{C})$ and a codomain $Y \in \mathsf{Obj}(\mathbb{C})$.

Maps may be composed using the $\circ$ operation, which is associative. For each object $X \in \mathsf{Obj}(\mathbb{C})$, there is an identity map $\mathsf{id}_X : X \to X$ which is neutral for the $\circ$ operation: for any map $f : X \to Y$, one has $f \circ \mathsf{id}_X = f = \mathsf{id}_Y \circ f$.

An object $I \in \mathsf{Obj}(\mathbb{C})$ is initial if for any object $X \in \mathsf{Obj}(\mathbb{C})$, there is a unique morphism $f : I \to X$ in $\mathsf{Hom}(\mathbb{C})$. Conversely, an object $F \in \mathsf{Obj}(\mathbb{C})$ is final if for any object $X \in \mathsf{Obj}(\mathbb{C})$, there is a unique morphism $f : X \to F$ in $\mathsf{Hom}(\mathbb{C})$.

Given two categories $\mathbb{C}$ and $\mathbb{D}$, a **functor** $F : \mathbb{C} \to \mathbb{D}$ consists of two mappings $\mathsf{Obj}(\mathbb{C}) \to \mathsf{Obj}(\mathbb{D})$ and $\mathsf{Hom}(\mathbb{C}) \to \mathsf{Hom}(\mathbb{D})$, both written F, such that:

- F preserves domains and codomains:
  if $f : X \to Y$ is in $\mathbb{C}$, $F(f) : F(X) \to F(Y)$ is in $\mathbb{D}$
- F preserves identities: $\forall X \in \mathbb{C}, F(\mathsf{id}_X) = \mathsf{id}_{F(X)}$
- F preserves composition:
  $\forall f : X \to Y$ and $g : Y \to Z$ in $\mathbb{C}$, $F(g \circ f) = F(g) \circ F(f)$ in $\mathbb{D}$.

Given two functors $F, G : \mathbb{C} \to \mathbb{D}$ from a category $\mathbb{C}$ to a category $\mathbb{D}$, a **natural transformation** $\varepsilon : F \Rightarrow G$ associates to any object $X \in \mathbb{C}$ a morphism $\varepsilon_X : F(X) \to G(X)$ in $\mathbb{D}$, called the component of ε at X, such that for every morphism $f : X \to Y$ in $\mathbb{C}$, we have $\varepsilon_Y \circ F(f) = G(f) \circ \varepsilon_X$.

2.2 Algebras and Coalgebras

Given an endofunctor $F : \mathbb{C} \to \mathbb{C}$ on a category $\mathbb{C}$, an F**-algebra** is defined by a carrier object $X \in \mathbb{C}$ and a morphism $\alpha : F(X) \to X$. In this categorical definition, F gives the signature of the algebra. For instance, with $\mathbf{1}$ denoting the singleton set $\{\star\}$, if we consider the functor $F = \mathbf{1} + \_$ which maps $X \mapsto \mathbf{1} + X$, the F-algebra $(\mathbb{N}, [0, \mathsf{succ}])$ is Peano's algebra of natural numbers, with the usual constant $0 : \mathbf{1} \to \mathbb{N}$ and constructor $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$.

Similarly, an F**-coalgebra** is defined by a carrier object $X \in \mathbb{C}$ and a morphism $\alpha : X \to F(X)$. In the common case where $\mathbb{C}$ is **Set**, the category of sets, the signature functor of an algebra describes operations for building elements of the carrier object. On the contrary, in a coalgebra, the signature functor describes operations for observing elements of the carrier objet. For instance, a Mealy machine can be described as a F-coalgebra $(S, \langle \mathsf{out}, \mathsf{next} \rangle)$ of the functor $F = (Out \times \_)^{In}$ with S, In and Out the sets of states, inputs and outputs.

2.3 Induction and Coinduction

An homomorphism of (co)algebras is a morphism from the carrier object of a (co)algebra to the carrier object of another (co)algebra which preserves the structure of the (co)algebras. On the following commutative diagrams, f is an homomorphism of algebras and g is an homomorphism of coalgebras:

$$\begin{tikzcd}
F(X) \arrow[r, "F(f)"] \arrow[d, "\alpha"'] & F(Y) \arrow[d, "\beta"'] \\
X \arrow[r, "f"'] & Y
\end{tikzcd}
\qquad
\begin{tikzcd}
Z \arrow[r, "g"] \arrow[d, "\delta"'] & U \arrow[d, "\gamma"'] \\
F(Z) \arrow[r, "F(g)"'] & F(U)
\end{tikzcd}$$

F-algebras and homomorphisms of algebras constitute a category $\mathbf{Alg}(F)$. Similarly, F-coalgebras and homomorphisms of coalgebras constitute a category $\mathbf{CoAlg}(F)$. If an initial algebra exists in $\mathbf{Alg}(F)$, it is unique, and its structure map is an isomorphism. The uniqueness of the homomorphism from an initial object to the other objects of a category is the key for defining morphisms by induction: giving the structure of an F-algebra (X, β) defines uniquely the homomorphism $f : I \to X$ from the initial F-algebra (I, α) to this algebra.

Conversely, if a final coalgebra exists in $\mathbf{CoAlg}(F)$, it is unique, and its structure map is an isomorphism. The uniqueness of the homomorphism from any object to a final object of a category is the key for defining morphisms by coinduction: giving the structure of an F-coalgebra (Y, δ) defines uniquely the morphism $f : Y \to F$ from this coalgebra to the final F-coalgebra (F, ω).

2.4 Monads

Monads [17] are a powerful abstraction for adding structure to objects. Given a category $\mathbb{C}$, a **monad** consists of an endofunctor $T : \mathbb{C} \to \mathbb{C}$ equipped with two natural transformations $\eta : \mathsf{id}_{\mathbb{C}} \Rightarrow T$ and $\mu : T^2 \Rightarrow T$ which satisfy the conditions $\mu \circ T\eta = \mu \circ \eta T = \mathsf{id}_{\mathbb{C}}$ and $\mu \circ T\mu = \mu \circ \mu T$:

$$\begin{tikzcd}
T^2 \arrow[dr, "\mu"'] & T \arrow[l, "T\eta"'] \arrow[r, "\eta T"] \arrow[d, "\mathsf{id}_{\mathbb{C}}"] & T^2 \arrow[dl, "\mu"] \\
& T &
\end{tikzcd}
\qquad
\begin{tikzcd}
T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"'] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"'] & T
\end{tikzcd}$$

η is called the *unit* of the monad. Its components map objects in $\mathbb{C}$ to their naturally structured counterpart. μ is the *product* of the monad. Its components map objects with two levels of structure to objects with only one level of structure. The first condition states that a doubly structured object $\eta_{T(X)}(t)$ built by η from a structured object t is flattened by μ to the same structured object as a structured object $T(\eta_X)(x)$ made of structured objects built by η. The second condition states that flattening two levels of structure can be made either by flattening the outer (with $\mu_{T(X)}$) or the inner (with $T(\mu_X)$) structure first.

Let us consider a monad built on the powerset functor $\mathcal{P} : \mathbf{Set} \to \mathbf{Set}$. We use it to model non-deterministic state machines by replacing the target state of a transition by a set of possible states. The component $\eta_S : S \to \mathcal{P}(S)$ of the unit of this monad has to build a set of states from a state. We can choose $\eta_S : \sigma \mapsto \{\sigma\}$. The component $\mu_S : \mathcal{P}(\mathcal{P}(S)) \to \mathcal{P}(S)$ of the product of the monad has to flatten a set of sets of states into a set of states. For a series of sets of states (s_i), $\forall i, s_i \in \mathcal{P}(S)$, we can choose $\mu_S : \{s_1 \ldots s_i \ldots\} \mapsto \cup s_i$.

Moreover, monads have also been used to represent many computation situations such as partiality, side-effects, exceptions, *etc* [21].

3 Transfer Functions and Components

In this section, we use the definition given by Barbosa in [1,19] to define components, *i.e.* as coalgebras of the **Set** endofunctor $T(Out \times \_)^{In}$ where In and Out are the sets of respectively input and output data and T is a monad. As we will see in Section 3.2, the interest of Barbosa's definition of components is that it is abstract enough to unify in a same framework a large family of formalisms classically used to specify state-based systems, such as Mealy machines, Labelled Transition Systems (LTS), Input-Output Labelled Transition Systems (IOLTS), etc. Similarly to Rutten's works in [11], we denote the behavior of a component by a transfer function.

3.1 Transfer Function

In the following, we note ω the least infinite ordinal, identified with the corresponding hereditarily transitive set.

Definition 1 (Dataflow). *A **dataflow** over a set of values A is a mapping $x : \omega \to A$. The set of all dataflows over A is noted A^ω.*

Transfer functions, which we use to describe the observable behavior of components, can be seen as dataflow transformers satisfying the causality condition in a standard framework [24], that is the output data at index n only depends on input data at indexes $0, \ldots, n$.

Definition 2 (Transfer function). *Let T be a monad. Let In and Out be two sets denoting, respectively, the input and output domains. A function $\mathcal{F} : In^\omega \longrightarrow Out^\omega$ is a **transfer function** if, and only if it is causal, that is:*

$$\forall n \in \omega, \forall x, y \in In^\omega, (\forall m, 0 \leq m \leq n, x(m) = y(m)) \Longrightarrow \mathcal{F}(x)(n) = \mathcal{F}(y)(n)$$

3.2 Components

Definition 3 (Components). *Let In and Out be two sets denoting, respectively, the input and output domains. Let T be a monad. A **component** $\mathcal{C}$ is a coalgebra (S, α) for the signature $\mathcal{H} = T(Out \times \_)^{In}$: **Set** $\to$ **Set** with a distinguished element s_0 denoting the initial state of the component $\mathcal{C}$.*

Example 1. We illustrate the notions and results previously mentioned with the simple example of a coffee machine $\mathcal{M}$ modeled by the transition diagram shown on Figure 1. The behavior of $\mathcal{M}$ is the following: from its initial state STDBY, when it receives a coin from the user, it goes into the READY state. Then, when the user presses the "coffee" button, it either serves a coffee to the user and goes to the STDBY state, or it fails to do so, refunds the user and goes to the FAILED state. The only escape from the FAILED state is to have a repair. In our framework, this machine is considered as a component $\mathcal{M} = (S, s_0, \alpha)$ over the signature[2] $\mathcal{P}_f(Out \times \_)^{In}$. The state space is

[2] $\mathcal{P}_f(X) = \{U \subseteq X | U \text{ is finite}\}$ is the finite powerset of X.

$S = \{\text{STDBY}, \text{READY}, \text{FAILED}\}$ and $s_0 = \text{STDBY}$. The sets of inputs and outputs are $In = \{\text{coin}, \text{coffee}, \text{repair}\}$ and $Out = \{\bot, \text{served}, \text{refund}\}$. Finally, the transition function $\alpha : S \longrightarrow \mathcal{P}_f\big(\{\bot, \text{served}, \text{refund}\} \times S\big)^{\{\text{coin},\text{coffee},\text{repair}\}}$ is defined as follows:

$$\begin{cases} \alpha(\text{STDBY})(\text{coin}) = \big\{(\bot, \text{READY})\big\} \\ \alpha(\text{READY})(\text{coffee}) = \big\{(\text{served}, \text{STDBY}), (\text{refund}, \text{FAILED})\big\} \\ \alpha(\text{FAILED})(\text{repair}) = \big\{(\bot, \text{STDBY})\big\} \end{cases}$$

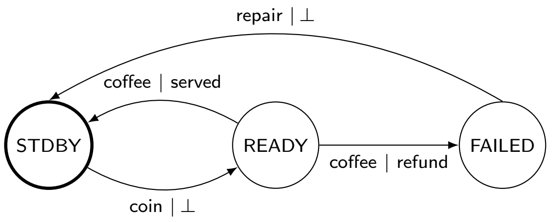


Fig. 1. Coffee machine

Definition 4 (Category of components). *Let $\mathcal{C}$ and $\mathcal{C}'$ be two components over $\mathcal{H} = T(Out \times \_)^{In}$. A* ***component morphism*** *$h : \mathcal{C} \to \mathcal{C}'$ is a coalgebra homomorphism $h : (S, \alpha) \to (S', \alpha')$ such that $h(s_0) = h(s'_0)$.*
We note $\mathbf{Cat}(\mathcal{H})$ *the* ***category of components*** *over $\mathcal{H}$.*

Using Definition 3 for components, we can unify in a same framework a large family of formalisms classically used to specify state-based systems such as Mealy machines, *LTS* and *IOLTS*. Hence, when T is the identity functor $\mathcal{I}d$, the resulting component is a Mealy machine. A Labelled Transition System is obtained by choosing $Out = \{\}$ and $In = Act$, a set of symbols standing for actions names, and the powerset functor $\mathcal{P}$ for T. Finally, with the powerset monad $\mathcal{P}$ for T, and with the additional property on the transition function $\alpha : S \longrightarrow \mathcal{P}(Out \times S)^{In}$:

$$\forall i \in In, \forall s \in S, (o, s') \in \alpha(s)(i) \implies \text{ either } i = \epsilon \text{ or } o = \epsilon$$

we obtain an $IOLTS$ (input and output are mutually exclusive).

3.3 Traces

To associate behaviors to components by their transfer function, we need to impose the supplementary condition on the monad T that there exists a natural transformation $\eta^{-1} : T \Longrightarrow \mathcal{P}$ where $\mathcal{P} : S \mapsto \mathcal{P}(S)$ is the powerset functor, such that: $\forall S \in \mathbf{Set}, \forall s \in S, \eta_S^{-1}(\eta_S(s)) = \{s\}$.

Most monads used to represent computation situations satisfy the above condition. For instance, for the monad $T : S \mapsto \mathcal{P}(S)$, η_S^{-1} is the identity on sets, while for the functor $T : S \mapsto S \cup \{\bot\}$, η_S^{-1} is the mapping that associates to $s \in S$ the singleton $\{s\}$ and the emptyset for $\bot$. The interest of η^{-1} is to allow the association of a set of transfer functions to a component (S, α) as its possible traces. Indeed, we need to "compute" for a sequence $x \in In^\omega$ all the outputs o after "performing" any sequence of states $(s_0, \ldots, s_k)$ such that s_j is obtained from s_{j-1} by $x(j-1)$. However, we do not know how to characterize s_j with respect to $\alpha(s_{j-1})(x(j-1))$. The problem is that nothing ensures that elements in $\alpha(s_{j-1})(x(j-1))$ are couples (output, state). Indeed, the monad T takes the product of a set of output Out and a set of states S and yields another set which may be different of the structure of $Out \times S$. The mapping $\eta^{-1}_{Out \times S}$ maps back to this structure.

In the following, we note $\eta^{-1}_{Out \times S}(\alpha(s)(i))_{|_1}$ (*resp.* $\eta^{-1}_{Out \times S}(\alpha(s)(i))_{|_2}$) the set composed of all first arguments (*resp.* second arguments) of couples in $\alpha(s)(i)$.

Definition 5 (Component traces)
Let $\mathcal{C}$ be a component over $\mathcal{H} = T(Out \times \_)^{In}$. The **Traces** *from a state s of $\mathcal{C}$ is the whole set of transfer functions $\mathcal{F}_s : In^\omega \to Out^\omega$ defined for every $x \in In^\omega$ such that there exists an infinite sequence of states $s_0, s_1, \ldots, s_k, \ldots \in S$ with $s_0 = s$ and satisfying: $\forall j \geq 1, s_j \in \eta^{-1}_{Out \times S}(\alpha(s_{j-1})(x(j-1)))_{|_2}$ and for every $k \in \omega$, $\mathcal{F}_s(x)(k) = o_k$* such that $(o_k, s_{k+1}) \in \eta^{-1}_{Out \times S}(\alpha(s_k)(x(k)))$
*Hence, $\mathcal{C}$***'s traces** *are the set of transfer functions $\mathcal{F}_{s_0}$ as defined above.*

In the context of our work, we are mainly interested by finite traces. Finite traces are finite sequences of couples (input|output) defined as follows :

Definition 6 (Component finite traces). *Let $\mathcal{F}_{s_0}$ be a trace of a component $\mathcal{C}$, let $n \in \mathbb{N}$. The* ***finite trace*** *of length n $\mathcal{F}_{{s_0}_{|n}}$ associated to $\mathcal{F}_{s_0}$ is the whole set of the finite sequence $\langle i_0|o_0, \ldots, i_n|o_n\rangle$ such that there exists $x \in In^\omega$ where for every j, $0 \leq j \leq n$, $x(j) = i_j$, and $\mathcal{F}_{s_0}(x(j)) = o_j$.*
Then, $Trace(\mathcal{C}) = \bigcup\limits_{\mathcal{F}_{s_0}} \bigcup\limits_{n \in \mathbb{N}} \mathcal{F}_{{s_0}_{|n}}$ defines the whole set of finite traces over $\mathcal{C}$.

4 Conformance Testing for Components

In this section, we examine how we can test the conformance of an implementation of a component to its specification. In order to compare the behavior of the implementation to the specification, we need to consider both as components over a same signature. However, the behavior of the implementation is unknown and can only be observed through its interface. We therefore need a conformance relation between what we can observe on the implementation and what the specification allows.

4.1 Conformance Relation

The specification *Spec* of a component is the formal description of its behavior given by a coalgebra over a signature $\mathcal{H} = T(Out \times \_)^{In}$. On the contrary, its implementation *SUT* (for *System under Test*) is an executable component, which is considered as a black box [3,25]. We interact with the implementation through its interface, by providing inputs to stimulate it and observing its behavior through its outputs.

The theory of conformance testing defines the conformance of an implementation to a specification thanks to conformance relations. Several kinds of relations have been proposed. For instance, the relations of *testing equivalence* and *preorders* [22,23] require the inclusion of trace sets. The relation *conf* [4] requires that the implementation behaves according to a specification, but allows behaviors on which the specification puts no constrain. The relation *ioconf* [26] is similar to *conf*, but distinguishes inputs from outputs. There are many other types of relations [15,20].

conf and *ioconf* have received most attention by the community of formal testing because they have shown their suitability for conformance testing. Since we are dealing with components with input and output, we choose *ioconf* and extend it to fit our framework. There are several extentions to *ioconf* according to both the underlying type of transition system and the aspect considered to be tested [8,9,13,5]. Recently, a denotational version of *ioconf* [27] was redefined in the Unifying Theories of Programming (UTP) [10].

Definition 7. *Let $\mathcal{C} = (S, s_0, \alpha)$ be a component. Let $tr = \langle i_0|o_0, \ldots, i_n|o_n\rangle$ be a finite trace over $\mathcal{C}$,* i.e. *an element of $Trace(\mathcal{C})$, and let s be a state of S. We have the two following definitions:*

– $$(\mathcal{C} \text{ after } tr) = \Big\{ s' \mid \exists s_1, \ldots, s_n \in S, \\ \forall j, 1 \leq j \leq n, (o_{j-1}, s_j) \in \eta^{-1}_{Out \times S}\big(\alpha(s_{j-1})(i_{(j-1)})\big), \\ \text{and } (o_n, s') \in \eta^{-1}_{Out \times S}\big(\alpha(s_n)(i_n)\big) \Big\}$$
is the set of reachable states from the state s_0 after executing tr

– $$Out_{\mathcal{C}}(s) = \bigcup_{i \in In} \Big(\{ o \mid \exists s' \in S, (o, s') \in \eta^{-1}_{Out \times S}\big(\alpha(s)(i)\big)\} \Big)$$
is the set of the possible outputs in s.

The set $Out_{\mathcal{C}}(s)$ can be extended to any set of states $S' \subseteq S$, we have :

$$Out_{\mathcal{C}}(S') = \bigcup_{s' \in S'} \big(Out_{\mathcal{C}}(s')\big)$$

These definitions allow us to define the *ioconf* relation in our framework:

Definition 8. *(ioconf) Let Spec and SUT be two components over the signature $T(Out \times \_)^{In}$. The **ioconf** relation is defined as follows :*

$$SUT \text{ ioconf } Spec \iff \begin{cases} \forall tr \in Trace(Spec), \\ Out_{SUT}(SUT \text{ after } tr) \subseteq Out_{Spec}(Spec \text{ after } tr) \end{cases}$$

We should note here that our *ioconf* definition covers all possible assumptions that must classically be made in conformance testing practice. For instance, it is always assumed that implementations are input enabled, that is, at any state, the implementation must produce an answer for all outputs. This assumption can naturally be expressed in our framework by considering the transition function α as total function.

4.2 Finite Computation Tree

In this section, we define the *finite computation tree* of a component, which captures all its finite computation paths:

Definition 9. *(Finite computation tree of component) Let (S, s_0, α) be a component over $T(Out \times \_)^{In}$. The* ***finite computation tree*** *of depth n of $\mathcal{C}$, noted $FCT(\mathcal{C}, n)$ is the coalgebra $(S_{FCT}, s^0_{FCT}, \alpha_{FCT})$ defined by :*

- *S_{FCT} is the whole set of $\mathcal{C}-$*paths*. A $\mathcal{C}-$*path *is defined by two finite sequences of states and inputs $(s_0, \dots, s_n)$ and $(i_0, \dots, i_{n-1})$ such that for every $j, 1 \leq j \leq n, s_j \in \eta^{-1}_{Out \times S}\big(\alpha(s_{j-1})(i_{j-1})\big)_{|_2}$*
- *s^0_{FCT} is the initial $\mathcal{C}-$*path *$\langle s_0, ()\rangle$*
- *α_{FCT} is the mapping which for every $\mathcal{C}-$*path *$\langle (s_0, \dots, s_n), (i_0, \dots, i_{n-1})\rangle$ and every input $i \in In$ associates $T(\Gamma)$ where Γ is the set:*

$$\Gamma = \Big\{\big(o, \langle (s_0, \dots, s_n, s'), (i_0, \dots, i_{n-1}, i)\rangle\big) \mid (o, s') \in \eta^{-1}_{Out \times S}\big(\alpha(s_n)(i)\big)\Big\}$$

In this definition, S_{FCT} is the set of the nodes of the tree. s^0_{FCT} is the root of the tree. Each node is represented by the unique $\mathcal{C}$-path $\langle (s_0, \dots, s_n), (i_0, \dots, i_{n-1})\rangle$ which leads to it from the root:

$$s_0 \xrightarrow{i_0} s_1 \xrightarrow{i_1} \dots \xrightarrow{i_{n-2}} s_{n-1} \xrightarrow{i_{n-1}} s_n$$

α_{FCT} gives, for each node p and for each input i, the set of nodes Γ that can be reached from p when the input i is submitted to the component.

4.3 Test Purpose

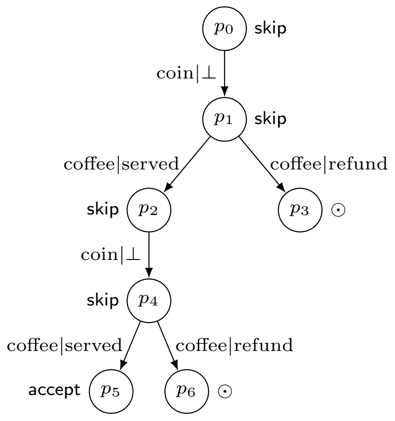
In order to guide the test derivation process, test purposes can be used. A test purpose is a description of the part of the specification that we want to test and for which test cases are to be generated. In [6] test purposes are described independently of the model of the specification. On the contrary, following [9], we prefer to describe test purposes by selecting the part of the specification that we want to explore. We therefore consider a test purpose as a tagged finite computation tree of the specification. The leaves of the FCT which correspond to paths that we want to test are tagged accept. All internal nodes on such paths are tagged skip, and all other nodes are tagged $\odot$.

Definition 10. *(Test Purpose) Let $FCT(\mathcal{C}, n)$ be the finite computation tree of depth n associated to a component $\mathcal{C}$. A* ***test purpose*** *TP for $\mathcal{C}$ is a mapping $TP : S_{FCT} \longrightarrow \{$accept, skip, $\odot\}$ such that:*

- *there exists a $\mathcal{C}-$*path *$p \in S_{FCT}$ such that $TP(p) =$* accept
- *if $TP(\langle (s_0, \dots, s_n), (i_0, \dots, i_{n-1})\rangle) =$* accept*, then:*
  for every $j, 1 \leq j \leq n-1, TP(\langle (s_0, \dots, s_j), (i_0, \dots, i_{j-1})\rangle) =$ skip

- $TP(\langle s_0, ()\rangle) = \mathsf{skip}$
- *if* $TP(\langle (s_0, \ldots, s_n), (i_0, \ldots, i_{n-1})\rangle) = \odot$, *then:*
  $TP(\langle (s_0, \ldots, s_n, s'_{n+1}, \ldots, s'_m), (i_0, \ldots, i_{n-1}, i'_n, \ldots, i'_{m-1})\rangle) = \odot$
  for all $m > n$ *and for all* $(s'_j)_{n<j\leq m}$ *and* $(i'_k)_{n\leq k<m}$

Example 2. Figure 2 gives a test purpose TP on the finite computation tree of depth 4 of the coffee machine $\mathcal{M}$ whose specification is shown on Figure 1. This test purpose allows us to ignore the behaviors of $\mathcal{M}$ related to failure and repair and to concentrate on its interaction with a user. When the machine fails and the user is refunded, we reach node p_3 or p_6 which are tagged with $\odot$. This indicates that we are not interested in further behavior from these nodes. p_5 is tagged with accept because it is a leaf which corresponds to an expected behavior. All nodes leading from the root p_0 to this node are tagged with skip because they are valid prefixes of p_5.



$$
\begin{aligned}
p_0 &= \langle \mathrm{STDBY}, ()\rangle \\
p_1 &= \langle (\mathrm{STDBY}, \mathrm{READY}), \\
&\quad \mathrm{coin}\rangle \\
p_2 &= \langle (\mathrm{STDBY}, \mathrm{READY}, \mathrm{STDBY}), \\
&\quad (\mathrm{coin}, \mathrm{coffee})\rangle \\
p_3 &= \langle (\mathrm{STDBY}, \mathrm{READY}, \mathrm{FAILED}), \\
&\quad (\mathrm{coin}, \mathrm{coffee})\rangle \\
p_4 &= \langle (\mathrm{STDBY}, \mathrm{READY}, \mathrm{STDBY}, \mathrm{READY}), \\
&\quad (\mathrm{coin}, \mathrm{coffee}, \mathrm{coin})\rangle \\
p_5 &= \langle (\mathrm{STDBY}, \mathrm{READY}, \mathrm{STDBY}, \mathrm{READY}, \mathrm{FAILED}), \\
&\quad (\mathrm{coin}, \mathrm{coffee}, \mathrm{coin}, \mathrm{coffee})\rangle \\
p_6 &= \langle (\mathrm{STDBY}, \mathrm{READY}, \mathrm{STDBY}, \mathrm{READY}, \mathrm{STDBY}), \\
&\quad (\mathrm{coin}, \mathrm{coffee}, \mathrm{coin}, \mathrm{coffee})\rangle
\end{aligned}
$$

Fig. 2. Test purpose of the coffee machine

In order to build a test purpose on a finite computation tree, we therefore choose the leaves of the tree which we accept as correct finite behaviors and we tag them with accept. We then tag every node which represents a prefix of an accepted behavior with skip. The other nodes, which lead to behaviors that we do not want to test, are tagged with $\odot$.

5 Test Generation Guided by Test Purposes

Similarly to [9], we propose an approach for test cases selection according to a test purpose. In order to test the conformance of the SUT to the specification, we start from the root of a test purpose, we choose a possible input i and submit it to the SUT. We observe the outputs o and compare them with the possible outputs in the finite computation tree. If the outputs do not match the specification, the verdict of the test is FAIL. Otherwise, if at least one of the nodes which can be reached with $i|o$ is tagged skip in the test purpose, the test goes on. If the nodes are tagged $\odot$, further behavior is not of interest, so the test is inconclusive (INCONC verdict). If one of the nodes is tagged accept, the test succeeds (PASS verdict). It may happen, due to the non-determinism of the specification, that the implementation behaved correctly, but we cannot determine if we reached an accept state or an $\odot$ state. This leads to a WeakPASS verdict.

5.1 Preliminaries

In this section, we introduce some notations and definitions that will be used in describing our algorithm for generating conformance tests for components.

As mentioned above, a test case is a sequence generated by a test purpose TP interacting with SUT. This is denoted by $[ev_0, ev_2, \ldots, ev_n][Verdict]$, where for all $i \in [0, \ldots, n], ev_i = i|o$ is an elementary input-output with $i \in In \cup \{\epsilon\}$ and $o \in Out \cup \{\epsilon\}$, and $Verdict \in \{FAIL, PASS, INCONC, WeakPASS\}$. We added the special symbol ϵ to the input and output actions to denote a stimulation of SUT without input and the absence of output for a stimulation. We note $stimobs(i|o)$ the output o from SUT when stimulating it with input i.

In order to compute the set of reachable states that lead to *accept* states after a given input-output sequence, we define a current set of states denoted by CS that contains a subset of the states of the test purpose. It is initialized to the initial state of TP. We also introduce three functions to help exploring TP by selecting paths that lead to *accept* states. $Next(CS, ev)$ is the set of directly reachable states from the current set of states CS after executing ev. $NextSkip(CS, ev)$ is the set of states in $Next(CS, ev)$ from which it is possible to reach accepting states, and $NextPass(CS, ev)$ is the set of states in $Next(CS, ev)$ which are labelled by *accept*.

Definition 11. *Let $TP : S_{FCT} \to \{accept, skip, \odot\}$ be a test purpose for a component $\mathcal{C}$, $ev = \langle i|o \rangle$ an event, and S' a subset of S_{FCT}:*

- $Next(S', ev) = \bigcup_{s' \in S'} (\{s \mid (o, s) \in \eta^{-1}_{Out \times S_{FCT}}(\alpha_{FCT}(s')(i))\})$,
- $NextSkip(S', ev) = Next(S', ev) \bigcap TP(S')_{|_{skip}}$,
- $NextPass(S', ev) = Next(S', ev) \bigcap TP(S')_{|_{accept}}$.

with $TP(S')_{|_{\mathsf{tag}}} = \{s' \in S' \mid TP(s') = \mathsf{tag}\}$

5.2 Inferences Rules

We define our test case generation algorithm as a set of inferences rules. Each rule states that under certain conditions on the next observation of output action from SUT or the next stimulation of SUT by an input action, the algorithm either performs an exploration of other states of TP, or stops by generating a verdict.

We structure these rules as $\frac{CS}{Results}$ $cond(ev)$, where CS is a set of current states, *Results* is either a set of current states or a verdict, and $cond(ev)$ is a set of conditions including $stimobs(ev)$. Each rule must be read as follows : *Given the current set of states CS, if* $cond(ev)$ *is verified, then the algorithm may achieve a step of execution, with ev as input-output elementary sequence.*

Our algorithm can be seen as an exploration of the finite computation tree starting from the initial state. It switches between sending stimuli to the implementation and waiting for output of the implementation according to the inference rules as long as a verdict is not reached. We distinguish two kinds of inference rules : *exploring* rules and *diagnosis* rules. The first kind, is applied to pursue the computation of the sequence as long as *Result* is a set of states. The second kind leads to a verdict and stops the algorithm.

Rule 0 : Initialization rule[3]: $\frac{}{\{s^0_{FCT}\}}$

Rule 1 : Exploration of other states : the emission o after a stimulation by i on the SUT is compatible with the test purpose but no accept is reached.

$$\frac{CS}{Next(CS, ev)}\ stimobs(ev),\ NextSkip(CS, ev) \neq \emptyset$$

Rule 2 : Generation of the verdict FAIL : the emission from the SUT is not expected with regards to the specification.

$$\frac{CS}{FAIL}\ stimobs(ev),\ Next(CS, ev) = \emptyset$$

Rule 3 : Generation of the verdict INCONC : the emission from the SUT is specified but not compatible with the test purpose.

$$\frac{CS}{INCONC} stimobs(ev), \begin{cases} Next(CS, ev) \neq \emptyset, \\ NextSkip(CS, ev) = NextPass(CS, ev) = \emptyset \end{cases}$$

Rule 4 : Generation of the verdict PASS : all next states directly reachable from the set of current set are *accept* ones.

$$\frac{CS}{PASS}\ stimobs(ev),\ NextPass(CS, ev) = Next(CS, ev),\ Next(CS, ev) \neq \emptyset$$

Rule 5 : Generation of the verdict WeakPASS : some of the next states are labelled by *accept*, but not all of them.

$$\frac{CS}{WeakPASS} stimobs(ev), \begin{cases} NextPass(CS, ev) \subset Next(CS, ev), \\ NextPass(CS, ev) \neq \emptyset \end{cases}$$

We should now note that each of these rules except rule 0 can be used in several ways according to the form of ev. When $ev = \epsilon|o$, o is produced spontaneously by SUT. When $ev = i|\epsilon$, the stimulation of SUT with i does not produce any output. Finally, when $ev = i|o$, o is produced by SUT when it is stimulated with i. These possibilities for ev therefore give rise to a generic algorithm that can be applied to a wide variety of state-based systems ([6,9,16]) by choosing the appropriate monad T and input and output sets.

[3] This rule is involved only once when starting the algorithm.

5.3 Properties

In order to state that, according to our algorithm, the non-existence of a FAIL verdict leads to conformance (correctness) and that any non-conformance is detected by a test case ending by a FAIL verdict (completeness), we denote by $\mathbb{CS}$ and $\mathbb{EV}$ respectively the whole set of current state sets and the whole set of input-output elementary sequences used during the application of the set of inference rules on an implementation SUT according to a test purpose TP. We then introduce a transition system whose states are the sets of current states and four special states labelled by the verdicts. Two states are linked by a transition labelled by an input-output elementary sequence. This transition system is formally defined as follows :

Definition 12. *Let TP be a test purpose for a specification Spec, let SUT be an implementation, let $\mathbb{CS}$ be the whole set of current state sets and let $\mathbb{EV}$ be the whole set of input-output elementary sequences. Then,* ***the execution of the test generation algorithm*** *on SUT according to TP denoted by $TS(TP, SUT)$ is the coalgebra (S_{TS}, α_{TS}) over the signature $(\_)^{\mathbb{EV}}$ defined by :*

- *$S_{TS} = \mathbb{CS} \cup \mathbb{V}\mathrm{erdict}$ where $\mathbb{V}\mathrm{erdict}$ is the set whose elements are FAIL, PASS, INCONC and WeakPASS,*
- *α_{TS} is the mapping which for every $CS \in \mathbb{CS}$ and for every $ev \in \mathbb{EV}$ is defined as follows :*

$$\alpha_{TS}(CS)(ev) = \begin{cases} Next(CS, ev) & \text{if } NextSkip(CS, ev) \neq \emptyset, NextPass(CS, ev) = \emptyset \\ FAIL & \text{if } Next(CS, ev) = \emptyset \\ INCONC & \text{if } NextSkip(CS, ev) = NextPass(CS, ev) = \emptyset \\ & \quad \text{and } Next(CS, ev) \neq \emptyset \\ PASS & \text{if } Next(CS, ev) = NextPass(CS, ev) \\ & \quad \text{and } Next(CS, ev) \neq \emptyset \\ WeakPASS & \text{if } NextPass(CS, ev) \subsetneq Next(CS, ev) \\ & \quad \text{and } NextPASS(CS, ev) \neq \emptyset \end{cases}$$

With this definition, test cases are sets of possible traces which can be observed during an execution of $TS(TP, SUT)$, and lead to a verdict state.

Definition 13. *Let $TS(TP, SUT) = (S_{TS}, \alpha_{TS})$ be the execution of the test generation algorithm on SUT according to TP. A* ***test case*** *for TP is a sequence $[ev_0, \ldots, ev_n][Verdict]$ for which there is a sequence of states $s_0, \ldots, s_n \in \mathbb{CS}$ with $\forall j, 0 \leq j < n, s_{j+1} = \alpha_{TS}(s_j)(ev_j)$, and there is a verdict state $Verdict \in \mathbb{V}\mathrm{erdict}$ such that $Verdict = \alpha_{TS}(s_n)(ev_n)$. We note $st(TP, SUT)$ the set of all possible test cases for TP.*

We can now introduce the notation:

$$vdt(TP, SUT) = \{Verdict \mid \exists ev_0 \ldots ev_n, [ev_0 \ldots ev_n | Verdict] \in st(TP, SUT)\}$$

Theorem 1. *(Correctness and completeness) For any specification Spec and any SUT:*

– ***Correctness:*** *If* SUT *conforms to Spec, for any test purpose* TP, $FAIL \notin vdt(TP, SUT)$.
– ***Completeness:*** *If* SUT *does not conform to Spec, there exists a test purpose* TP *such that* $FAIL \in vdt(TP, SUT)$.

6 Conclusion

In this paper, we have presented a coalgebraic model, a conformance relation between implementations and specifications, and a test generation algorithm for component based systems. This work relies on previous works done in [1,19] for defining software components as coalgebras, and in [9] for defining our test generation algorithm.

The formalism used in this paper to specify both the specification and system the under test is abstract enough to subsume most state-based formalisms. Hence, the conformance theory defined here over this formalism is *de facto* a generalization of standard theories found for different state-based formalisms.

The ability of this framework to model and generate tests for components is a first step toward the testing of complex (software) systems, made from a huge number of components that interact altogether. This will require the definition of integration operators to combine the behavior of components. It should then allow us to check whether an implementation made of conforming components combined with integration operators is conform to its specification.

In order to fit the required format of the paper, we omitted some details (detailed explanations, theorems of existence of final coalgebras, proofs of all theorems) which are available in an extended version of this paper [14].

References

1. Barbosa, L.S.: Towards a calculus of state-based software components. Journal of Universal Computer Science 9(8), 891–909 (2003)
2. Barr, M., Wells, C. (eds.): Category theory for computing science, 2nd edn. Prentice Hall International (UK) Ltd., Hertfordshire (1995)
3. Bernot, G.: Testing against formal specifications: A theoretical view. In: TAPSOFT 1991: Proc. of the Intl. Joint Conference on Theory and Practice of Software Development, London, UK, vol. 2, pp. 99–119. Springer, Heidelberg (1991)
4. Brinksma, E.: A theory for the derivation of tests. In: Proc. 8th Int. Conf. Protocol Specification, Testing, and Verification (PSTV VIII), pp. 63–74 (1988)
5. Briones, L., Brinksma, E.: A test generation framework for quiescent real-time systems. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 64–78. Springer, Heidelberg (2005)
6. Jéron, T., Jard, C.: TGV: theory, principles and algorithms. International Journal on Software Tools for Technology Transfer 7(4), 297–315 (2005)
7. Fiadeiro, J.L.: Categories for Software Engineering. Springer, Heidelberg (2004)
8. Frantzen, L., Tretmans, J., Willemse, T.A.C.: A Symbolic Framework for Model-Based Testing. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) FATES 2006 and RV 2006. LNCS, vol. 4262, pp. 40–54. Springer, Heidelberg (2006)

9. Gaston, C., Le Gall, P., Rapin, N., Touil, A.: Symbolic execution techniques for test purpose definition. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, pp. 1–18. Springer, Heidelberg (2006)
10. Jifeng, H., Hoare, C.A.R.: Unifying theories of programming. In: Orlowska, E., Szalas, A. (eds.) RelMiCS, pp. 97–99 (1998)
11. Hansen, H.H., Costa, D., Rutten, J.J.M.M.: Synthesis of mealy machines using derivatives. Electr. Notes Theor. Comput. Sci. (ENTCS) 164(1), 27–45 (2006)
12. Hardebolle, C., Boulanger, F.: Exploring multi-paradigm modeling techniques. SIMULATION: Transactions of the Society for Modeling and Simulation International 85(11/12), 688–708 (2009)
13. Heerink, A.W., Tretmans, G.J.: Refusal testing for classes of transition systems with inputs and outputs. In: Mizuno, T., Shiratori, N., Higashino, T., Togashi, A. (eds.) Proceedings of the IFIP TC6 WG6.1 Joint Intl. Conf. on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE X) and Protocol Specification, Testing and Verification (PSTV XVII). IFIP Conference Proceedings, vol. 107, pp. 23–38. Chapman & Hall, Boca Raton (1997)
14. Kanso, B., Aiguier, M., Boulanger, F., Touil, A.: Testing of abstract components. Internal Report 2010-05-28-DI-FBO, Supélec (2010), http://wwwdi.supelec.fr/
15. Langerak, R.: A testing theory for LOTOS using deadlock detection. In: Brinksma, E., Scollo, G., Vissers, C.A. (eds.) Protocol Specification, Testing and Verification (PSTV), pp. 87–98. North-Holland, Amsterdam (1989)
16. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines—a survey. Proceedings of the IEEE 84(8) (August 1996)
17. Mac Lane, S.: Categories for the Working Mathematician. Graduate Texts in Mathematics, vol. 5. Springer, Heidelberg (1971)
18. Mealy, G.H.: A method for synthesizing sequentiel circuits. Bell Systems Techn. Jour. (1955)
19. Meng, S., Barbosa, L.S.: Components as coalgebras: the refinement dimension. Theor. Comput. Sci. (TCS) 351(2), 276–294 (2006)
20. Milner, R.: Communication and concurrency. Prentice-Hall, Inc., Upper Saddle River (1989)
21. Moggi, E.: Notions of computation and monads. Information and Computation 93, 55–92 (1991)
22. De Nicola, R., Hennessy, M.C.B.: Testing equivalences for processes. Theoretical Computer Science (TCS) 34(1-2), 83–133 (1984)
23. Phillips, I.: Refusal testing. Theor. Comput. Sci. 50(3), 241–284 (1987)
24. Sontag, E.D.: Mathematical control theory: deterministic finite dimensional systems, 2nd edn. Springer, New York (1998)
25. Tretmans, J.: A formal approach to conformance testing. In: Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems VI, pp. 257–276. North-Holland Publishing Co., Amsterdam (1994)
26. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software - Concepts and Tools 17(3), 103–120 (1996)
27. Weilghofer, M., Aichernig, B.: Unifying input output conformance. In: Butterfield, A. (ed.) UTP 2008. LNCS, vol. 5713, pp. 181–201. Springer, Heidelberg (2010)

Scalable Distributed Concolic Testing: A Case Study on a Flash Storage Platform[⋆]

Yunho Kim, Moonzoo Kim, and Nam Dang

CS Dept. KAIST
Daejeon, South Korea
{kimyunho,moonzoo}@kaist.ac.kr,
paddy@kaist.ac.kr

Abstract. Flash memory has become a virtually indispensable component for mobile devices in today's information society. However, conventional testing methods often fail to detect hidden bugs in flash file systems due to the difficulties involved in creating effective test cases. In contrast, the approach of model checking guarantees a complete analysis, but only on a limited scale. In the previous work, the authors applied concolic testing to the multi-sector read operation of a Samsung flash storage platform as a trade-off between the aforementioned two methods.

This paper describes our continuing efforts to develop an effective and efficient verification framework for flash file systems. We developed a scalable distributed concolic algorithm that utilizes a large number of computing nodes. This new concolic algorithm can alleviate the limitations of the concolic approach caused by heavy computational cost. We applied the distributed concolic technique to the multi-sector read operation of a Samsung flash storage platform and compared the empirical results with results obtained with the original concolic algorithm.

1 Introduction

On the strengths of characteristics such as low power consumption and strong resistance to physical shock, flash memory has become a crucial component for mobile devices. Accordingly, in order for mobile devices to operate successfully, it is imperative that the flash storage platform software (e.g., file system, flash translation layer, and low-level device driver) operates correctly. However, conventional testing methods often fail to detect hidden bugs in flash storage platform software, since it is difficult to create effective test cases (i.e., test cases that provide a check of all possible execution scenarios generated from complex flash storage platform software). Thus, the current industrial practice of manual testing does not achieve high reliability or provide cost-effective testing. As another testing approach, randomized testing can save human effort for test case generation. However, it does not achieve high reliability, because random input

⋆ This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST)/National Research Foundation of Korea(NRF) (Grant 2010-0001727) and the MKE(Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) support program supervised by NIPA(National IT Industry Promotion Agency) (NIPA-2010-(C1090-1031-0001)).

data does not necessarily guarantee high coverage of a target program. These deficiencies of conventional testing incur significant overhead to manufacturers. In spite of the importance of flash memory, however, little research work has been conducted to formally analyze flash storage platforms. In addition, most of such work [10,7] has focused on the specifications of file system design, not real implementation.

In the previous work [12], the authors applied *concolic* (CONCrete + symbOLIC) testing [16,8,4] (also known as dynamic symbolic execution [18] or automated white-box fuzzing [9]) to the multi-sector read operation (MSR) of the Samsung OneNAND flash storage platform [15] and tested all possible execution paths in an *automatic and exhaustive* manner. We used CREST [3] (an open source concolic testing tool for C) in the experiments and confirmed that concolic testing was effective to detect bugs. However, CREST consumed a large amount of time to analyze all possible execution paths, which is not acceptable in an industrial setting. For example, it took more than three hours to test a MSR with a small explicit environment consisting of 5 physical units and 6 logical sectors, which generated 2.8×10^6 test cases in total. Although concolic testing effectively detects bugs through the full path coverage, the required heavy computational cost prohibits the use of concolic testing in real world applications.

This paper describes our continuing efforts to develop an effective and efficient verification framework for flash file systems by alleviating the limitations caused by heavy computational cost. One solution is to develop a *scalable distributed concolic algorithm* that can utilize a large number of computing nodes with high efficiency. Thus far, most of automated formal verification techniques such as model checking have suffered heavy computational costs. Consequently, this heavy overhead often prevents practitioners from adopting these valuable techniques. The concolic approach is a suitable technique to exploit the benefits of parallel computing. We modified the original concolic algorithm to utilize multiple computing nodes in a distributed manner so as to reduce time cost significantly. In addition, this distributed concolic algorithm is scalable to utilize a large number of computing nodes, achieving linear speedup with an increasing number of computing nodes. We applied this distributed concolic technique on the multi-sector read operation (MSR) of a Samsung flash storage platform with 16 computing nodes. This paper reports experimental results obtained with the new concolic approach and compares them with the results derived with the original concolic algorithm to demonstrate the former's performance gain and scalability.

The organization of this paper is as follows. Section 2 explains the original concolic testing algorithm. Section 3 describes the distributed concolic algorithm. Section 4 overviews the multisector-read (MSR) function of the Samsung flash storage platform. Section 5 presents the experimental results obtained by applying the distributed concolic algorithm to MSR. Section 6 concludes the paper along with directions for future work.

2 Original Concolic Testing Algorithm

This section presents the original concolic testing algorithm [16,8,4]. Concolic testing executes a target program both concretely and symbolically [14,19] at the same time. Concolic testing proceeds via the following five steps:

1. *Instrumentation*
   A target C program is statically instrumented with probes, which record symbolic path conditions (PCs) from a concrete execution path when the target program is executed. Note that PCs correspond to conditional statements (i.e., `if`) in the target program.
2. *Concrete execution*
   The instrumented C program is executed with given input values and the concrete execution part of the concolic execution constitutes the normal execution of the program. For the first execution of the target program, the initial inputs are assigned with random values. For the second execution and onward, input values are obtained from step 5.
3. *Obtaining a symbolic path formula ϕ_i*
   The symbolic execution part of the concolic execution collects symbolic path conditions over the symbolic input values at each branch point encountered along the concrete execution path. Whenever each statement s of the target program is executed, a corresponding probe inserted at s updates the symbolic map of symbolic variables if s is an assignment statement, or collects a corresponding symbolic path condition pc, if s is a branch statement. Thus, a complete symbolic path formula ϕ_i of the ith execution is the conjunction of all PCs $pc_1, pc_2, ...pc_n$ where pc_j is executed earlier than pc_{j+1} for all $1 \leq j < n$.
4. *Generating a symbolic path formula ϕ'_i for the next input values*
   Given a symbolic path formula ϕ_i obtained in Step 3, to obtain the next input values, ϕ'_i is generated by negating the path condition pc_j (initially $j = n$) and removing

```
Input:
$path$: a sequence of PCs executed in the previous execution
$neg\_limit$: a position of a PC in $path$ beyond which PCs should not be negated
Output:
a set of generated test cases (i.e., $I$'s of line 7)
1  $Concolic(path, neg\_limit)$ {
2  $j =| path |$ ;
3  while $j >= neg\_limit$ do
4      // $\phi$ is a symbolic path formula of $path$
5      // $pc_k$ is $k$th path condition of $path$ and $pc_1$ is executed first
6      $\phi = pc_1 \wedge ... \wedge pc_{j-1} \wedge \neg pc_j$ ;
7      $I = SMT\_Solver(\phi)$ // returns NULL if $\phi$ is unsatisfiable
8      if $I$ is not NULL then
9          $path'$ = execute a target program on $I$ ;
10         $Concolic(path', j + 1)$;
11     end
12     $j = j - 1$;
13 end
14 }
```

Algorithm 1. Original concolic algorithm

the subsequent PC (i.e., $pc_{j+1}, ...pc_n$) of ϕ_i. If ϕ'_i is unsatisfiable, another path condition pc_{j-1} is negated and the subsequent PCs are removed, until a satisfiable path formula is found. If there are no further available new paths, the algorithm terminates.

5. *Selecting the next input values*
   A constraint solver such as a Satisfiability Modulo Theory (SMT) solver [17] generates a model that satisfies ϕ'_i. This model decides concrete next input values and the entire concolic testing procedure iterates from Step 2 again with these input values.

Algorithm 1 describes the original concolic algorithm in detail, which corresponds to Step 2 to Step 5. Algorithm 1 negates all PCs of a given path one by one in descending order (see line 3 to line 13) and new paths ($path'$ in line 9) are analyzed recursively (see line 10). To prevent redundant analysis of a given path, subsequent recursive $Concolic()$ negates PCs of $path'$ up to neg_limit th PC (i.e., only $pc_{|path'|}, pc_{|path'|-1}, ..., pc_{|neg\_limit|}$ of $path'$ are negated one by one). Note that this concolic algorithm operates in a similar manner to the depth first order (DFS) traversal of the execution tree of a target program.

3 Distributed Concolic Algorithm

This section describes a *distributed* concolic algorithm that can utilize a large number of computing nodes. The main concept underlying the new algorithm is based on the feature that symbolic path formulas in the loop (line 3 to line 13 of Algorithm 1) of the original concolic algorithm are analyzed *independently*. Therefore, in order to analyze these symbolic path formulas in a distributed manner, Algorithm 2 generates and stores symbolic path formulas in $queue_{pf}$ (line 15) without analyzing these symbolic path formulas recursively (line 10 of Algorithm 1). If $queue_{pf}$ is empty (exiting the loop of line 5 to line 25) and there are no more paths to analyze in all distributed nodes, the algorithm terminates (line 31). Otherwise, the current node requests a symbolic path formula from another node n' (line 27) and receives a symbolic path formula from n' (line 28). The received symbolic path formula is then added into $queue_{pf}$ (line 29) and the algorithm continues from line 5 again. If the current node receives a request for symbolic path formulas (line 17), it sends one from $queue_{pf}$ (lines 19 and 20) immediately as long as $queue_{pf}$ is not empty.[1]

Note that communication between nodes occurs only when $queue_{pf}$ is empty. Since $queue_{pf}$ is non-empty for most of the analysis time, the number of communications is small compared to the number of analyzed symbolic path formulas. In addition, the communicated message contains only one symbolic path formula, whose size is small (proportional to the length of the corresponding execution path). Furthermore, this algorithm is not affected by the complexity and/or characteristics of a target program. Therefore, Algorithm 2 is scalable to utilize a large number of computing nodes without performance degradation.

[1] In a real implementation, there is a server to coordinate communications between computing nodes; this is not described in this paper for the sake of providing a simple description.

```
Input:
$orig\_path$: a sequence of PCs executed in the previous execution
Output:
a set of generated test cases (i.e., $I$'s of line 12)
1  $DstrConcolic(orig\_path)$ {
2  $queue_{pf} = \emptyset$; // queue containing symbolic path formulas
3  Add $(orig\_path, 1)$ to $queue_{pf}$;
4  repeat
5      while $queue_{pf}$ is not empty do
6          Remove $(path, neg\_limit)$ from $queue_{pf}$;
7          $j =| path |$;
8          while $j >= neg\_limit$ do
9              // $\phi$ is a symbolic path formula of $path$
10             // $pc_k$ is $k$th path condition of $path$ and $pc_1$ is executed first
11             $\phi = pc_1 \wedge ... \wedge pc_{j-1} \wedge \neg pc_j$ ;
12             $I = SMT\_Solver(\phi)$; // returns NULL if $\phi$ is unsatisfiable
13             if $I$ is not NULL then
14                 $path'$ = execute the target program on $I$;
15                 Add $(path', j+1)$ to $queue_{pf}$;
16             end
17             if there is a request for a symbolic path formula from other node $n$ then
18                 if $queue_{pf}$ is not empty then
19                     Remove $(path'', neg\_limit'')$ from $queue_{pf}$;
20                     Send $(path'', neg\_limit'')$ to $n$;
21                 end
22             end
23             $j = j - 1$;
24         end
25     end
26     if there are uncovered paths in any distributed node then
27         Send a request for a symbolic path formula to $n'$ whose $queue_{pf}$ is not empty;
28         Receive $(path, neg\_limit)$ from $n'$;
29         Add $(path, neg\_limit)$ to $queue_{pf}$;
30     end
31 until all execution paths are covered;
32 }
```

Algorithm 2. Distributed concolic algorithm

4 Overview of Multi-sector Read Operation

Unified storage platform (USP) is a software solution to operate a Samsung flash memory device [15]. USP allows applications to store and retrieve data on flash memory through a file system. USP contains a flash translation layer (FTL) through which data and programs in the flash memory device are accessed. The FTL consists of three layers - a sector translation layer (STL), a block management layer (BML), and a low-level device driver layer (LLD). Generic I/O requests from applications are fulfilled through the file system, STL, BML, and LLD, in order. MSR resides in STL.[2]

4.1 Overview of Sector Translation Layer (STL)

A NAND flash device consists of a set of *pages*, which are grouped into *blocks*. A *unit* can be equal to a block or multiple blocks. Each page contains a set of *sectors*.

When new data is written to flash memory, rather than overwriting old data directly, the data is written on empty physical sectors and the physical sectors that contain the old data are marked as invalid. Since the empty physical sectors may reside in separate physical units, one logical unit (LU) containing data is mapped to a linked list of physical units (PU). STL manages this mapping from logical sectors (LS) to physical sectors (PS). This mapping information is stored in a sector allocation map (SAM), which returns the corresponding PS offset from a given LS offset. Each PU has its own SAM.

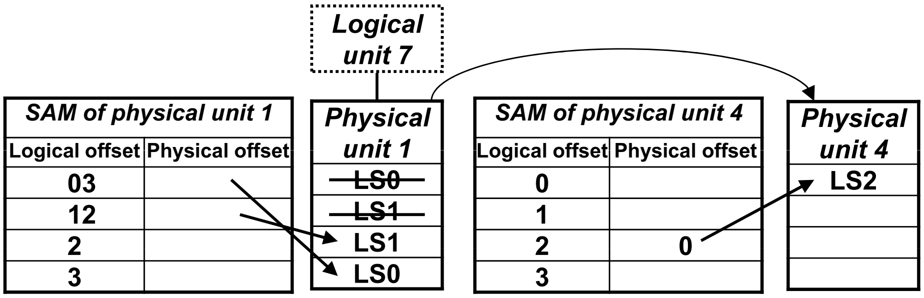


Fig. 1. Mapping from logical sectors to physical sectors

Figure 1 illustrates a mapping from logical sectors to physical sectors where 1 unit consists of 1 block and 1 block contains 4 pages, each of which consists of 1 sector. Suppose that a user writes LS0 of LU7. An empty physical unit PU1 is then assigned to LU7, and LS0 is written into PS0 of PU1 (SAM1[0]=0). The user continues to write LS1 of LU7, and LS1 is subsequently stored into PS1 of PU1 (SAM1[1]=1). The user then updates LS1 and LS0 in order, which results in SAM1[1]=2 and SAM1[0]=3. Finally, the user adds LS2 of LU7, which adds a new physical unit PU4 to LU7 and yields SAM4[2]=0.

4.2 Multi-sector Read Operation

USP provides a mechanism to simultaneously read as many multiple sectors as possible in order to improve the reading speed. The core logic of this mechanism is implemented in a single function in STL. Due to the non-trivial traversal of data structures for logical-to-physical sector mapping (see Section 4.1), the function for MSR is 157 lines long and

[2] This section is taken from [11].

highly complex, having 4-level nested loops. Figure 2 describes simplified pseudo code of these 4-level nested loops. The outermost loop iterates over LUs of data (line 2-18) until the numScts amount of the logical sectors are read completely. The second outermost loop iterates until the LSes of the current LU are completely read (line 5-16). The third loop iterates over PUs mapped to the current LU (line 7-15). The innermost loop identifies consecutive PSes that contain consecutive LSes in the current PU (line 8-11). This loop calculates conScts and offset, which indicate the number of such consecutive PSes and the starting offset of these PSes, respectively. Once conScts and offset are obtained, BML_READ rapidly reads these consecutive PSes as a whole (line 12).

```
01:curLU = LU0;
02:while(numScts > 0) {
03:  readScts = # of sectors to read in the current LU
04:  numScts -= readScts;
05:  while(readScts > 0 ) {
06:    curPU = LU->firstPU;
07:    while(curPU != NULL ) {
08:      while(...) {
09:        conScts=# of the consecutive PSes to read in curPU
10:        offse =the starting offset of the consecutive PSes
11:      }
12:      BML_READ(curPU, offset, conScts);
13:      readScts = readScts - conScts;
14:      curPU = curPU->next;
15:    }
16:  }
17:  curLU = curLU->next;
18:}
```

Fig. 2. Loop structures of MSR

For example, suppose that the data is “ABCDEF” and each unit consists of four sectors and PU0, PU1, and PU2 are mapped to LU0 (“ABCD”) in order and PU3 and PU4 are mapped to LU1 (“EF”) in order, as depicted in Figure 3(a). Initially, MSR accesses SAM0 to find which PS of PU0 contains LS0(‘A’). It then finds SAM0[0]=1 and reads PS1 of PU0. Since SAM0[1] is empty (i.e., PU0 does not have LS1(‘B’)), MSR moves to the next PU, which is PU1. For PU1, MSR accesses SAM1 and finds that LS1(‘B’) and LS2(‘C’) are stored in PS1 and PS2 of PU1 consecutively. Thus, MSR reads PS1 and PS2 of PU1 altogether through BML_READ and continues its reading operation.

The requirement property for MSR is that the content of the read buffer should be equal to the original data in the flash memory when MSR finishes reading, as given by assert($\forall i$.LS[i]==buf[i]) inserted at the end of MSR.[3]

[3] [13] describes a systematic method to identify this test oracle for MSR.

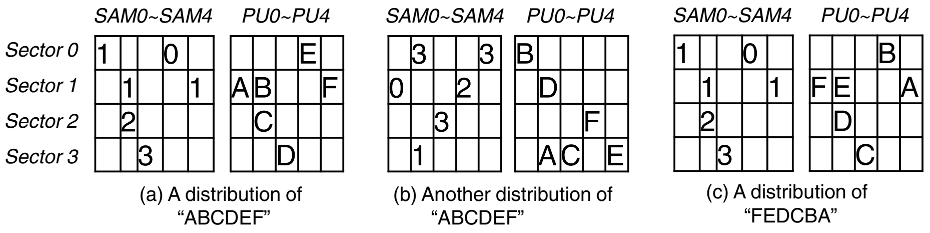

SAM0~SAM4
PU0~PU4
Sector 0
Sector 1
Sector 2
Sector 3
1 0
1 1
2
3
E
A B F
C
D
3 3
0 2
3
1
B
D
F
A C E
1 0
1 1
2
3
B
F E A
D
C
(a) A distribution of "ABCDEF"
(b) Another distribution of "ABCDEF"
(c) A distribution of "FEDCBA"

Fig. 3. Possible distributions of data "ABCDEF" and "FEDCBA" to physical sectors

In these analysis tasks, we assume that each sector is 1 byte long and each unit has four sectors. Also, we assume that data is a fixed string of distinct characters (e.g., "ABCDE" if we assume that data is 5 sectors long, and "ABCDEF" if we assume that data is 6 sectors long). We apply this data abstraction, since the values of logical sectors should not affect the reading operations of MSR, but the distribution of logical sectors into physical sectors does. For example, for the same data "ABCDEF", the reading operations of MSR are different for Figure 3(a) and Figure 3(b), since they have different SAM configurations (i.e., different distributions of "ABCDEF"). However, for "FEDCBA" in Figure 3(c), which has the same SAM configuration as the data shown in Figure 3(a), MSR operates in exactly same manner as for Figure 3(a). Thus, if MSR reads "ABCDEF" in Figure 3(a) correctly, MSR reads "FEDCBA" in Figure 3(c) correctly too.

In addition, we assume that data occupies 2 logical units. The number of possible distribution cases for l LSes and n physical units, where $5 \leq l \leq 8$ and $n \geq 2$, increases exponentially in terms of both n and l, and can be obtained by

$$\sum_{i=1}^{n-1} \left({}_{(4\times i)}C_4 \times 4! \right) \times \left({}_{(4\times(n-i))}C_{(l-4)} \times (l-4)! \right)$$

For example, if a flash has 1000 physical units with data occupying 6 LSes, there exist a total of 3.9×10^{22} different distributions of the data. Table 1 shows the total number of possible cases for 5 to 8 logical sectors and various numbers of physical units, respectively, according to the above formula.

Table 1. Total number of the distribution cases

PUs	4	5	6	7	8
$l = 5$	61248	290304	9.8×10^5	2.7×10^6	6.4×10^6
$l = 6$	239808	1416960	5.8×10^6	1.9×10^7	5.1×10^7
$l = 7$	8.8×10^5	7.3×10^6	3.9×10^7	1.5×10^8	5.0×10^8
$l = 8$	3.4×10^6	4.2×10^7	2.9×10^8	1.4×10^9	5.6×10^9

MSR has the characteristics of a control-oriented program (4-level nested loops) and a data-oriented program (large data structure consisting of SAMs and PUs) at the same time, although the values of PSes are not explicitly manipulated. As seen from Figure 3, the execution paths of MSR depend on the values of SAMs and the order of PUs linked to LU. In other words, MSR operates deterministically, once the configuration of the SAMs and PUs is fixed.

5 Case Study on Paralleized Concolic Testing of the Flash Storage Platform

In this section, we describe a series of experiments for testing the multisector read (MSR) operation of the unified storage platform (USP) for a Samsung OneNAND flash memory [15]. Also, we compare the empirical results of applying distributed concolic testing with the results of the original concolic testing [12].

Our goal is to investigate the distributed concolic algorithm, focusing on its performance improvement and scalability when applied to MSR. We thus pose the following research questions.

- **RQ1**: How does the distributed concolic algorithm improve the speed of concolic testing the MSR code?
- **RQ2**: How does the distributed concolic algorithm achieve scalability when applied to the MSR code?

5.1 Environment Model

MSR assumes that logical data are randomly written on PUs and the corresponding SAMs record the actual location of each LS. The writing is, however, subject to several constraint rules; the following are some of the representative rules. The last two rules can be enforced by the constraints in Figure 4.

1. One PU is mapped to at most one LU.
2. If the i_{th} LS is written in the k_{th} sector of the j_{th} PU, then the $(i \; mod \; m)_{th}$ offset of the j_{th} SAM is valid and indicates the PS number k, where m is the number of sectors per unit (4 in our experiments).
3. The PS number of the i_{th} LS must be written in *only* one of the $(i \; mod \; m)_{th}$ offsets of the SAM tables for the PUs mapped to the $\lfloor \frac{i}{m} \rfloor_{th}$ LU.

To enforce such constraints on test cases, a test driver/environment model generates valid (i.e., satisfying the environment constraints) test cases *explicitly* by selecting a PU and its sector to contain the l th logical sector (`PU[i].sect[j]=LS[l]`) and setting the corresponding SAM accordingly (`SAM[i].offset[l]=j`).

For example, Figure 3(a) represents the following distribution case:

- LS[0]=‘A’, LS[1]=‘B’, LS[2]=‘C’, LS[3]=‘D’, LS[4]=‘E’, and LS[5]=‘F’.
- PU[0].sect[1]=‘A’, PU[1].sect[1]=‘B’, PU[1].sect[2]=‘C’, PU[2].sect[3]=‘D’, PU[3].sect[0]=‘E’, and PU[4].sect[1] = ‘F’.

$$\begin{aligned}\forall i,j,k\ (LS[i] = PU[j].sect[k] \rightarrow (&SAM[j].valid[i\ mod\ m] = true\\ &\&\ SAM[j].offset[i\ mod\ m] = k\\ &\&\ \forall p.(SAM[p].valid[i\ mod\ m] = false)\\ &\text{where } p \neq j \text{ and } PU[p] \text{ is mapped to} \lfloor \frac{i}{m} \rfloor_{th}\ LU))\end{aligned}$$

Fig. 4. Environment constraints for MSR

- SAM[0].valid[0]=true, SAM[1].valid[1]=true, SAM[1].valid[2]=true, SAM[2].valid[3]=true, SAM[3].valid[0]=true, and SAM[4].valid[1]=true (all other validity flags of the SAMs are false).
- SAM[0].offset[0]=1, SAM[1].offset[1]=1, SAM[1].offset[2]=2, SAM[2].offset[3] =3, SAM[3].offset[0]=0, and SAM[4].offset[1]=1.

Thus, the environment contraints for $i = 2, j = 1$, and $k = 2$ are satisfied as follows:

$$\begin{aligned}LS[2] = PU[1].sect[2] \rightarrow (&SAM[1].valid[2\ mod\ 4] = true\\ &\&\ SAM[1].offset[2\ mod\ 4] = 2\\ &\&\ SAM[0].valid[2\ mod\ 4] = false\\ &\&\ SAM[2].valid[2\ mod\ 4] = false\\ &\&\ SAM[3].valid[2\ mod\ 4] = false)\end{aligned}$$

5.2 Test Setup for the Experiment

All experiments were performed on 64 bit Fedora Linux 9 equipped with a 3.6 GHz Intel Core2Duo processor and 16 gigabytes of memory. We utilized 16 computing nodes connected with a gigabit ethernet switch. We implemented Algorithm 2 in the open source concolic testing tool CREST [2]. However, since the CREST project is in its early stage, CREST has several limitations such as lack of support for dereferencing of pointers and array index variables in the symbolic analysis. Consequently, the target MSR code was modified to use an array representation of the SAMs and PUs. We used CREST 0.1.1 (with DFS search option), gcc 4.3.0, Yices 1.0.24 [5], which is an SMT solver used as an internal constraint solver by CREST for solving symbolic path formulas. Although CREST does not correctly test programs with non-linear arithmetic, we could apply CREST to MSR successfully, because MSR contains only linear integer arithmetic.

To evaluate the effectiveness of parallelized concolic testing (i.e., bug detecting capability), we applied *mutation analysis* [1] by injecting the following three types of frequently occuring bugs (i.e. mutation operators), as we did in our previous study [12]. The injected bugs are as follows:

1. *Off-by-1 bugs*

   - b_{11}: `while(numScts>`**0**`)` of the outermost loop (line 2 of Figure 2) to `while(numScts>`**1**`)`
   - b_{12}: `while(readScts>`**0**`)` of the second outermost loop (line 5 of Figure 2) to `while(readScts>`**1**`)`
   - b_{13}: `for(i=0;i<conScts; i++)` of `BML_READ()` (line 12 of Figure 2) to `for(i=0;i<conScts`**-1**`;i++)`
2. *Invalid condition bugs*

   - b_{21}: `if(SAM[i].offset[j]`**!=**`0xFF)` in the third outermost loop to `if(SAM[i].offset[j]`**==**`0xFF)`
   - b_{22}: `readScts=((4-j)>numScts)?numScts:4-j` in the innermost loop to `readScts=((4-j)<numScts)?numScts:4-j`
   - b_{23}: `if((firstOffset+nScts)`**==**`SAM[i].offset[j])` in the innermost loop to `if((firstOffset+nScts)`**!=**`SAM[i].offset[j])`

3. *Missing statement bugs*

   - b_{31}: missing `nScts=1` in the second outermost loop
   - b_{32}: missing `nReadScts--` in the second outermost loop
   - b_{33}: missing `nLun++` corresponding the line 17 of Figure 2

To evaluate the efficiency of parallelized concolic testing, we measured the total testing time to cover all possible execution paths.

5.3 Experimental Results

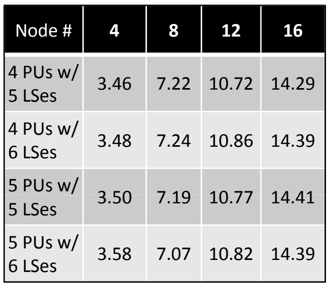
Regarding RQ1: How does the distributed concolic algorithm improve the speed of concolic testing the MSR code. We performed 4 series of experiments with 4 to 5 PUs with 5 to 6 LSes and with 1, 4, 8, 12, and 16 computing nodes. The total numbers of test cases generated and corresponding time costs are reported in Table 2. For example, 1.1×10^5 test cases were generated for MSR with 4 PUs w/ 5 LSes (see the last column of Table 2). 1 computing node took 643 seconds to generate 1.1×10^5 test cases for the experiment with 4 PUs and 5 LSes. However, 4, 8, 12, and 16 nodes took only 186, 89, 60, and 45 seconds for the same experiment, respectively (see the second row of

Table 2. Total number of generated test cases and time costs (seconds)

	1	4	8	12	16	# test cases
4 PUs w/ 5 LSes	643	186	89	60	45	1.1×10^5
4 PUs w/ 6 LSes	3194	919	441	294	222	5.3×10^5
5 PUs w/ 5 LSes	3242	927	451	301	225	4.9×10^5
5 PUs w/ 6 LSes	19225	5369	2718	1777	1336	2.8×10^6

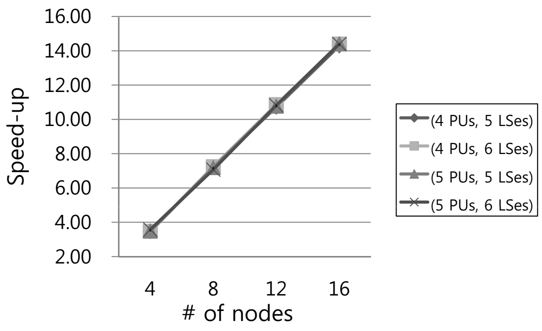
Table 2). Therefore, compared to the original concolic testing (the second column of Table 2), the distributed concolic testing reduced time cost significantly.

In a similar manner, we can analyze speedup achieved by the distributed concolic algorithm. Figure 5 illustrates speedup results with a different number of computing nodes. For example, with an environment containing 5 PUs and 6 LSes (the last row of the table in Figure 5), 4 computing nodes completed all testing 3.58 times faster than 1 computing node ($3.58 = \frac{19225}{5369}$). Similarly, 8, 12, and 16 computing nodes with the same environment completed concolic testing 7.07, 10.82, 14.39 times faster, respectively.



Node #	4	8	12	16
4 PUs w/ 5 LSes	3.46	7.22	10.72	14.29
4 PUs w/ 6 LSes	3.48	7.24	10.86	14.39
5 PUs w/ 5 LSes	3.50	7.19	10.77	14.41
5 PUs w/ 6 LSes	3.58	7.07	10.82	14.39

(a) Table of speed-up ratios for different numbers of nodes



(b) Graph of speed-up ratios for different numbers of nodes

Fig. 5. Speed-up ratios for different numbers of nodes

Regarding the effectiveness of bug detection, the results of the distributed concolic algorithm are the same as the ones of the original concolic testing. The distributed concolic testing detected violations of the requirement property (`assert(` $\forall i$.`LS[` i`]==buf[`i`])`) due to the all 9 bugs in a few seconds. Thus, the distributed concolic testing reduces time costs significantly without loss of effectiveness compared to the original concolic testing.

Regarding RQ2:How does the distributed concolic algorithm achieve scalability when applied to the MSR code. As shown in the table of Figure 5, the efficiency of parallelism ($\frac{\text{speedup}}{\text{\# of nodes}}$) is almost 90% regardless of the number of nodes. For example, MSR with an environment consisting of 5 PUs and 6 LSes showed almost the same parallelism efficiency for different numbers of nodes (see the last row of the table in Figure 5 where $\frac{3.58}{4} \approx \frac{7.07}{8} \approx \frac{10.82}{12} \approx \frac{14.39}{16} \approx 90\%$). Furthermore, as shown in the graph of Figure 5, the distributed concolic algorithm achieved almost identical parallelism efficiency regardless of the environment configurations. In other words, 4 PUs with 5 LSes, 4 PUs with 6 LSes, 5 PUs with 5 LSes, and 5 PUs with 6 LSes achieve almost identical parallelism efficiency. This observation indicates that the performance of the distributed concolic algorithm is *not* affected by the complexity of a target program either, which is another advantage of our distributed concolic algorithm.

Table 3. Overhead due to the distributed concolic testing

	Waiting time (%)				Communication (%)				Waiting + Communication (%)			
	4	8	12	16	4	8	12	16	4	8	12	16
4 PUs w/ 5 LSes	2.54	2.63	2.65	2.79	1.09	1.22	1.84	1.72	3.64	3.85	4.49	4.51
4 PUs w/ 6 LSes	2.46	2.56	2.58	2.61	1.31	1.42	1.45	1.78	3.77	3.98	4.03	4.39
5 PUs w/ 5 LSes	2.23	2.28	2.30	2.33	1.49	1.56	1.53	1.21	3.72	3.85	3.83	3.54
5 PUs w/ 6 LSes	2.19	2.11	2.16	2.17	1.15	3.55	1.50	1.46	3.35	5.67	3.66	3.64

Table 4. Numbers of iterations performed by nodes

# nodes	(PUs, LSes)	Node ID																Avg	Stdev
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16		
4	(4,5)	27.4	31.5	27.0	25.3													27.8	2.6
$(\times 10^3)$	(4,6)	122.6	136.6	124.2	151.4													133.7	13.4
	(5,5)	117.3	121.1	134.8	116.0													122.3	8.6
	(5,6)	771.0	690.0	645.4	670.7													694.3	54.3
8	(4,5)	13.4	14.0	13.0	13.6	14.6	13.5	15.1	14.0									14.0	0.7
$(\times 10^3)$	(4,6)	63.5	63.8	63.6	69.6	67.6	62.6	71.5	72.6									66.9	4.0
	(5,5)	59.8	68.9	58.5	59.0	59.3	60.4	62.3	60.9									61.1	3.4
	(5,6)	335.3	334.5	355.4	332.7	399.1	347.0	332.9	340.2									347.1	22.5
12	(4,5)	8.9	9.2	10.4	8.9	9.2	8.6	9.0	8.8	9.7	10.5	9.2	9.0					9.3	0.6
$(\times 10^3)$	(4,6)	42.7	42.9	48.4	42.3	49.3	42.8	45.5	43.0	42.4	41.5	44.2	49.7					44.6	3.0
	(5,5)	42.9	39.2	40.8	38.7	40.1	39.3	42.4	40.3	39.5	40.3	39.6	46.2					40.8	2.1
	(5,6)	216.2	239.0	219.8	221.6	220.7	239.8	233.0	249.8	272.9	222.0	222.6	219.5					231.4	16.7
16	(4,5)	7.2	7.3	7.4	6.9	7.0	7.2	6.8	6.6	6.3	6.6	6.6	7.4	7.2	7.7	6.4	6.8	7.0	0.4
$(\times 10^3)$	(4,6)	32.2	31.8	34.0	32.7	34.1	33.4	32.2	37.3	31.3	37.3	32.6	32.1	32.7	32.1	32.5	36.5	33.4	2.0
	(5,5)	30.1	29.4	31.8	29.9	32.1	32.4	29.4	33.3	29.7	29.1	31.6	28.9	29.1	30.4	30.9	31.2	30.6	1.4
	(5,6)	167.2	207.5	167.3	189.6	168.5	167.1	166.5	167.9	170.5	176.0	174.4	174.0	165.3	180.3	171.0	163.9	173.6	11.2

We expect that a high parallelism efficiency can be achieved even with a large number of computing nodes (saying hundreds of thousand computing nodes of cloud computing), since there is little dependency among computing nodes. Communication between two nodes occurs only when one of the nodes has completely generated and analyzed all possible subsequent symbolic paths from a given symbolic path (i.e., when $queue_{pf}$ is empty). Thus, each node can concentrate on its own computational task with little waiting/blocking for other nodes. For the similar reason, communication costs caused by the distributed concolic algorithm are also insignificant.

Table 3 describes overhead caused by the distributed concolic algorithm. For example, with 4 computing nodes, MSR with an environment model of 4 PUs and 5 LSes spent 2.54% of the whole execution time (on average) by waiting/idling until it received a symbolic path formula from another node (between line 27 and line 28 of Algorithm 2). In addition, MSR with the same environment spent 1.09% of the whole execution time for socket communication. Thus, these two overheads of the distributed concolic algorithm constitute 3.64% of the whole execution time. The remaining 6% ($\approx 10\% - 3.64\%$) of overhead is caused by the increased complexity of the distributed concolic algorithm such as maintaining $queue_{pf}$ and handling communication at user process level, etc. Considering that the current implementation of the distributed concolic algorithm is not optimized, this overhead can be reduced further.

Another evidence of the scalability of the algorithm can be found in Table 4, which describes the numbers of iterations (test case generations) performed by the computing nodes. For example, with MSR with an environment model of 4 PUs and 5 LSes, nodes 1, 2, 3, and 4 performed 27.4×10^3, 31.5×10^3, 27.0×10^3, and 25.3×10^3 iterations, respectively. The numbers of iterations for different nodes are roughly similar, which means that an equal amount of work was assigned to each node, which improves global utilization of computing nodes. Note that time cost for each iteration may vary depending on the length of a corresponding symbolic path formula. Measured time costs of nodes (not shown in this paper) are even more identical.

6 Conclusion and Future Work

We have developed a distributed concolic algorithm which can reduce time cost by utilizing a large number of computing nodes. Furthermore, we have demonstrated the improved performance of the algorithm through an industrial case study on the multisector-read operation of a Samsung flash storage platform. We applied the distributed concolic algorithm to the MSR code and analyzed the approach empirically. In this case study, the distributed concolic algorithm achieved an order of magnitude faster testing speed compared to the original concolic algorithm while maintaining the effectiveness of bug detection capability. Although these experiments were performed on only 16 computing nodes, we could observe that the algorithm has good characteristics of scalable distributed algorithms such as linear speedup with an increasing number of nodes, little blocking time, and nominal communication overheads. Therefore, we expect that the distributed algorithm can alleviate problems caused by heavy computational costs in a large degree.

We plan to apply the distributed concolic algorithm to target programs on 10,000 nodes of the Amazon EC2 platform [6] to demonstrate the scalability of the algorithm in a concrete manner. Furthermore, this experiment can suggest a promising direction of fighting the state space explosion problem of automated verification techniques. Finally, we will develop a new concolic algorithm for branch coverage for more practical applications in an industrial setting.

References

1. Andrews, J.H., Briand, L.C., Labiche, Y.: Is mutation an appropriate tool for testing experiments? In: International Conference on Software Engineering, ICSE (2005)
2. Burnim, J.: CREST - automatic test generation tool for C, `http://code.google.com/p/crest/`
3. Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. Technical Report UCB/EECS-2008-123, EECS Department, University of California, Berkeley (September 2008)
4. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Operating System Design and Implementation, OSDI (2008)
5. Dutertre, B., Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
6. Amazon Elastic Compute Cloud (Amazon EC2), `http://aws.amazon.com/ec2/`

7. Ferreira, M.A., Silva, S.S., Oliveira, J.N.: Verifying Intel flash file system core specification. In: 4th VDM-Overture Workshop (2008)
8. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Programming Language Design and Implementation, PLDI (2005)
9. Godefroid, P., Levin, M.Y., Molnar, D.: Automated whitebox fuzz testing. In: Network and Distributed Systems Security (2008)
10. Kang, E., Jackson, D.: Formal modeling and analysis of a flash filesystem in Alloy. In: Abstract state machines, B and Z (2008)
11. Kim, M., Choi, Y., Kim, Y., Kim, H.: Formal verification of a flash memory device driver - an experience report. In: Spin Workshop (2008)
12. Kim, M., Kim, Y.: Concolic testing of the multi-sector read operation for flash memory file system. In: Oliveira, M.V.M., Woodcock, J. (eds.) SBMF 2009. LNCS, vol. 5902, pp. 251–265. Springer, Heidelberg (2009)
13. Kim, M., Kim, Y., Kim, H.: Unit testing of flash memory device driver through a SAT-based model checker. In: Automated Software Engineering (ASE) (September 2008)
14. King, J.C.: Symbolic execution and program testing. Communications of the ACM 19(7) (1976)
15. Samsung OneNAND fusion memory, `http://www.samsung.com/global/business/semiconductor/products/fusionmemory/Products_OneNAND.html`
16. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: European Software Engineering Conference/Foundations of Software Engineering, ESEC/FSE (2005)
17. SMT-LIB: The satisfiability module theories library, `http://combination.cs.uiowa.edu/smtlib/`
18. Tillmann, N., Schulte, W.: Parameterized unit tests. In: European Software Engineering Conference/Foundations of Software Engineering, ESEC/FSE (2005)
19. Visser, W., Pasareanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. In: International Symposium on Software Testing and Analysis, ISSTA (2004)

Analyzing a Formal Specification of Mondex Using Model Checking

Reng Zeng and Xudong He

School of Computing and Information Sciences,
Florida International University,
Miami, Florida 33199, USA
{rzeng001,hex}@cis.fiu.edu

Abstract. Mondex, an electronic purse, is the first pilot project of the software verification Grand Challenge to establish the correctness of software. Several research groups around the world have applied different formal methods in specifying and analyzing the Mondex since 2006. In this paper, we present a method to analyze the SAM specification of Mondex using model checking. Our specification uses SAM that integrates high level Petri nets and temporal logic. Our analysis method translates the SAM Mondex specification into a behavior preserving PROMELA program and uses SPIN to model check the resulting PROMELA program. Our results and experiences are discussed, which contributes to the world wide effort in developing a verified software repository.

Keywords: Mondex, Grand Challenge, Model Checking, High Level Petri Net, SAM.

1 Introduction

In recent years, both the Computing Research Association in the U.S. and the UK Computing Research Committee proposed a set of grand challenges in computing sciences. One common grand challenge proposed by the above organizations is on developing dependable software systems [CRC08] [Woo06]. The Mondex smart card, an electronic purse, was chosen as the 1st pilot project in 2006. The objectives were to demonstrate how research groups can collaborate and compete in scientific experiments, and to generate artifacts to populate the verified software repository [VSR07].

Mondex is a payment system, an electronic purse system, based on smart card technology, which offers an alternative to paying cash for goods and services, allowing person-to-person payment. In 1999, Mondex was awarded a security rating of ITSEC Level E6 [WSC$^+$07] - the highest possible rating achievable in ITSEC (Information Technology Security Evaluation Criteria).

During the development of Mondex, Z was used to specify and to prove the correctness of the Mondex design [SCW00]. Since no network access was required for transaction, it demanded critically high security level on each Mondex purse itself. Z Specifications were used to prove the following security properties of Mondex:

1. no value may be created in the system: the sum of all the purses' balances does not increase; and
2. all value must be accounted for in the system: the sum of all purses' balances and lost components does not change.

The security properties were proved manually, which was evaluated by a third party group, and a sanitised version of the proof was published in 2000. The proof has critically helped Mondex be granted ITSEC security level 6 , the highest level.

In [ZLH08], we presented a formal specification of Mondex in SAM [HD02], a formal software architecture model integrating high-level Petri nets and temporal logic. In this paper, we present a way using model checking to analyze the formal specification of Mondex in SAM. This formal specification and verification contributes to the world wide effort on developing a verified software repository.

2 Specifying Mondex in SAM

A formal specification of Mondex in SAM was developed in [ZLH08]. This section gives a brief SAM specification of the abstract model.

2.1 SAM

SAM [HD02], an architectural description model based on Petri nets and temporal logic, is well-suited for modeling distributed systems. A SAM specification is hierarchical consisting of multiple compositions. Each composition may contain multiple elements. Each element $C = (B, S)$ has a behavior model B (modeled in a high level Petri net [HM05]), and a property specification S (defined by a temporal logic formula). An element is correctly designed if the behavior model B satisfies the property specification S, denoted by $B \models S$. The correctness of a SAM architecture description is defined recursively from the correctness of all elements.

A high level Petri net B is a tuple $(P, T, F, Spec, \varphi, R, \mathcal{L}, M_0)$ where (P, T, F) is the net structure, *Spec* is the underlying algebraic specification that defines the static semantics of net elements, and $(\varphi, R, \mathcal{L}, M_0)$ is the net inscription that maps net elements to terms in the algebraic specification. φ associates each place in P with a type in *Spec*. R associates each transition in T with a boolean term in *Spec*. M_0 is the initial marking which associates each place in P with type respecting ground terms in *Spec*. We assume that the reader has some knowledge of Petri nets and temporal logic, and thus omit their formal definitions, which can be found in [HD02]. In the sequel, we simply use Petri nets to refer to high level Petri nets.

2.2 The Abstract Model

In the Z Specification of Mondex [SCW00], *ether* is used to model the communication channel. Messages between purses could be lost, and also could be read by third parties as there may be somebody eavesdropping, so *ether* is designed as lossy and public, all request messages are initially in *ether*. Each purse interacts with card reader via a connector, contact or contactless. Each purse accepts input from card reader, which could be either an initial request in *ether*, or the message sent out by another purse. Each purse produces an output to *ether*.

Accordingly in the SAM model of Mondex, two places, *msg_in* and *msg_out*, are used to model the communication channel, shown in Fig. 1, in which *msg_in* contains tokens for input messages, and *msg_out* contains tokens for output messages. All request messages are initially in *msg_in*, and each purse accepts input messages from *msg_in*. For output messages, each purse sends them to *msg_out*. All messages in *msg_in* comes from *ether*, and all messages in *msg_out* goes to *ether*.

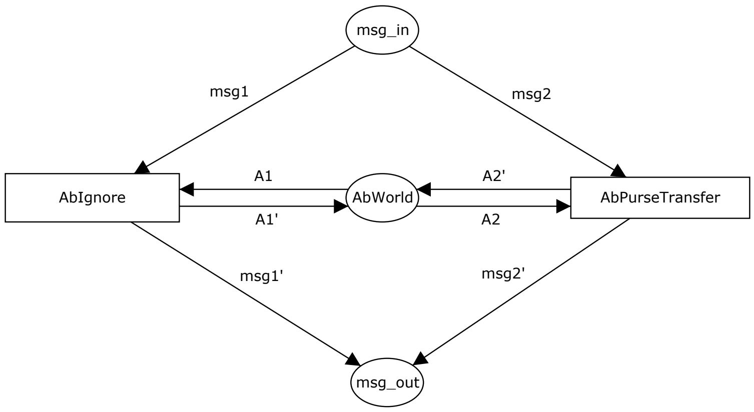


Fig. 1. The Abstract Model

The abstract model has only one atomic operation to transfer money from the paying purse to the receiving one. It corresponds to transition *AbPurseTransfer* in Fig. 1. Transition *AbIgnore* is introduced in Fig. 1 to handle invalid messages.

The whole world of abstract purses is modeled using the power set of purses, *AbWorld*.

The net inscription for the abstract model is given below, which defines the types of places, constraints of transitions, and the initial marking. The definition of arc labels are omitted since they are self evident in Fig. 1.

The Types of Places. The type of *msg_in* contains information of operations and parameters. An operation can be *aNullIn* or *transfer*, and parameters provide transferring details including the name of *from* side (paying party), the name of *to* side (receiving party), and the value to transfer. The type of *msg_in* is thus defined as below.

$$OP = \{aNullIn, transfer\} \tag{1}$$
$$\varphi(msg\_in) = OP \times string \times string \times \mathbb{N} \tag{2}$$

The type of *AbWorld* is the power set of purses, in which each purse has 3 fields, the first field defines the name of each purse, the second one defines balance and the third one defines lost value.

$$\varphi(AbWorld) = \mathbb{P}(string \times \mathbb{N} \times \mathbb{N}) \tag{3}$$

The type of *msg_out* is modeled as *aNullOut*.

$$\varphi(msg\_out) = \{aNullOut\} \tag{4}$$

The Constraints of Transitions. The precondition of transition *AbIgnore* tests that the message *msg1* contains operation *aNullIn*, and its postcondition keeps *AbWorld* unchanged.

$$R(AbIgnore) = (msg1[1] = aNullIn) \wedge (A1' = A1) \tag{5}$$

For transition *AbPurseTransfer*, its inputs are a message from *msg_in* denoted by *msg2* and all abstract purses from *AbWorld* denoted by *A2*. R (AbPurseTransfer) is the constraint for transition *AbPurseTransfer*, which assures the purse *m* is the *from* side and purse *n* is the *to* side, and *m* is not the same purse as *n*. It also updates the balance in abstract world.

$$\begin{aligned} R(AbPurseTransfer) = {} & (msg2\,[1] = transfer) \wedge \\ & \exists\,(m \in A2, n \in A2) \bullet (\\ & \quad m[1] = msg2[2] \wedge n[1] = msg2[3] \\ & \quad \wedge\, msg2[2] \neq msg2[3] \\ & \quad \wedge\, A2' = A2 \setminus \{m, n\} \cup \\ & \qquad \{(m[1], (m[2] - msg2[4]), m[3]), \\ & \qquad (n[1], (n[2] + msg2[4]), n[3]) \\ & \qquad \} \\ &) \end{aligned} \tag{6}$$

The Initial Marking. Any permissible initial marking can be provided. To demonstrate the dynamic behavior of our specification, the following initial marking is used.

$$\begin{aligned} M_0(msg\_in) &= \{(transfer, 1, 2, 50)\} \\ M_0(msg\_out) &= \{\} \\ M_0(AbWorld) &= \{\{(P1, 100, 0), (P2, 200, 0), (P3, 150, 0)\}\} \end{aligned} \tag{7}$$

3 Analyzing the Specification in Sam

Model checking is an automatic and efficient method for analyzing finite state systems, which is well suited for this Sam specification. In Sam, model checking is to ensure $B \models S$, that is the behavior model B satisfies the property specification S. The behavior model B uses high level Petri net, which employs sets and power sets as the type of places. The property specification S uses linear temporal logic. Spin uses Promela as its input language to model the behavior, and uses linear temporal logic to specify the properties. In order to use Spin for model checking a Sam specification, the behavior model B is translated to Promela code, and the property specification S remains the same. Translation between formal models are often useful, various issues with regard to formal model translation were discussed in [KG02].

3.1 Spin and Promela

Spin [Hol03] is a well known model checking tool used in the verification of finite state systems. Promela, as the input language of Spin, consists of processes, channels, and variables. For the channels, there are operations to fetch messages from them randomly or first-in-first-out, and to fetch the messages with desired field value. It is also possible to test the existence of desired messages in channels while not changing anything.

Specifically, single question mark "?" is a Promela operator that returns the first message in the channel, double question mark "??" is a Promela operator that returns the first matched message in the channel, "[...]" is a Promela testing operator returning true or false, while does not block the execution and does not retrieve messages from the channel, and "<...>" is a Promela channel poll operator which retrieves a message without removing it from the channel if a desired message exists in the channel. There is a predefined unary function in Promela called *eval* to turn an expression into a value. "!" is a Promela operator that sends a message to the channel.

3.2 Rules to Translate High Level Petri Net to Promela

This section introduces the rules to translate a high level Petri net to Promela, with the abstract model of Mondex (Fig. 1) as the example, however, the rules are also applied to the concrete model of Mondex for model checking discussed in Section 4. Before discussing the details of rules, we outline the translation by explaining the mapping from a high level Petri net to Promela code, as shown in Table 1.

Without the loss of generality, we assume all the types in a Petri net model are directly definable in Promela in this paper, since we can always make a type conversion before the translation.

Step 1. Define places as channels. Each place is translated into a Promela channel; and tokens are translated into messages. Specifically, let $p \in P$ be a

Table 1. Outline of mapping relationships from Petri Nets to PROMELA

Petri Nets	Description
Places	Places contain tokens, while in PROMELA channel contains messages, thus places are translated into channels.
Transitions	Each transition is translated into a PROMELA inline function.
Transition constraints	The constraints for each transition have 2 parts: precondition and postcondition.
Initial markings	The initial marking is translated to initial messages in the channel.

place in a Petri net with type $\varphi(p) = s_1, s_2, ..., s_n$, we define a bounded channel in PROMELA as follows.

```
#define Bound_p const
chan type_p = [Bound_p] of {s1, s2, ..., sn};
```

where *const* is a user defined positive integer value. Line 4 in Appendix 4 is a translation example of place AbWorld in Fig. 1 with type defined in Formula 3. Line 6 in Appendix 4 is a translation example of place msg_in in Fig. 1 with type defined in Formula 2. The types of places AbWorld and msg_in are different, yet both of them are translated into channels, the difference between types of places is addressed in following Step 2 and Step 3 by using different PROMELA operators.

Step 2. Define the inline functions for the precondition of a transition The inline function works like a usual preprocessor macro. It is introduced here to offer better translation structure and facilitate automated translation.

Formally, for each transition $t \in T$ with constraint:

$$R(t) = PreCond(t) \wedge PostCond(t) \tag{8}$$

where $PreCond(t)$ is the precondition of transition t and $PostCond(t)$ is the postcondition of transition t. $R(t)$ contains basic relational expressions connected through logical conjunction $\wedge$ or logical disjunction $\vee$, in which $PreCond(t)$ contains only variables on input arcs and $PostCond(t)$ contains variables on output arcs with or without variables on input arcs. Let $v \in \mathcal{L}(p, t)$ denote a simple variable in case v does not have a power set type. Let $v \in S$, $S \in \mathcal{L}(p, t)$, S has a power set type, v denotes a quantified variable. We assume the first field of either simple variables or quantified variables be the key field, and for those variables v containing only one field, each reference of v is viewed as $v[1]$.

We use the constraint (Formula 6) of transition *AbPurseTransfer* as an example in this section, in which the part above the line is the precondition and the part below the line is the postcondition.

We define an inline function to check the enabledness of the precondition of each transition. First, we define a boolean variable *t_is_enabled* to store the truth value of the checking for transition *t*, with initialized value false, refer to Step 5 below. Second, for the fields of each simple variable or quantified variable, we define corresponding variables. Let v be the name of simple variable or quantified variable containing n fields, $TYPE(i)$ be the type of i^{th} field, we define $TYPE(i)\ v\_fieldi$; for $i \in 2..n$. For example, we define Line 27-28 in Appendix 4 for Formula 6.

Table 2. General Mapping from basic relational expressions in the precondition of each transition in a Petri Net to PROMELA Expressions

Basic Relational Expression	PROMELA Expressions
$v[1] = Exp$ where $v \in \mathcal{L}(p,t)$, $p \in P$, $t \in T$ and v is a simple variable containing n fields, Exp does not contain any first field.	$type\_p$? $<eval(Exp),$ $v\_field2, v\_field3, .., v\_fieldn$ $>$
$\exists(v \in S) \centerdot (v[1] = Exp)$ where $v \in S$, $S \in \mathcal{L}(p,t)$, $p \in P$, $t \in T$ and v is a quantified variable containing n fields, Exp does not contain the first field of any quantified variable.	$type\_p$?? $[eval(Exp),$ $v\_field2, v\_field3, .., v\_fieldn$ $]$

Table 2 gives the general mapping for basic relational expressions connected through logical conjunction $\wedge$ or logical disjunction $\vee$. We use single question marks for simple variables such that messages in the channel are retrieved in FIFO order, and we use double question marks for quantified variables since existential quantification implies a search throughout the whole power set. We use "[...]" to test the existence of messages in case a truth value is needed for *if* statement and the matched message does not require a copy, and we use "<...>" to make a guard statement for *if* statement in PROMELA, so that only in case there is a desired message, the statements after the guard statement are executed and the matched message is copied, for example, in Appendix 4, Line 29 is a guard statement for Line 60, where the matched message is copied to msg2_field2 to msg2_field4 for each field..

Table 3 gives the mapping for the precondition in Formula 6.

Line 26-37 in Appendix 4 is the resulting PROMELA code.

Step 3. Define the inline function for the postcondition of a transition
For each transition, once its precondition is met, it can fire. This section introduces the rules to define an inline function for the postcondition of a transition firing.

In the rules for the precondition, we test enabledness without moving any tokens, thus as part of the postcondition we move tokens through input arcs. For a simple variable v on an input arc a message from the head of channel obtained

Table 3. Mapping from the precondition in Formula 6 to PROMELA Expressions

Basic Relational Expression	PROMELA Expressions
$msg2[1] = transfer$	$type\_msg\_in? < eval(transfer), msg2\_field2,$ $msg2\_field3, msg2\_field4 >$
$m[1] = msg2[2]$	$type\_AbWorld??[eval(msg2\_field2), m\_field2,$ $m\_field3]$
$n[1] = msg2[3]$	$type\_AbWorld??[eval(msg2\_field3), n\_field2,$ $n\_field3]$
$msg2[2] \neq msg2[3]$	$msg2\_field2 \,!= msg2\_field3$

Table 4. General Mapping from basic relational expressions in the postcondition of each transition in a Petri Net to PROMELA Expressions

Basic Relational Expression	PROMELA Expressions
$v[1] = Exp$ where $v \in \mathcal{L}(p,t)$ $, p \in P,\ t \in T$, and v is a simple variable containing n fields.	$type\_p \ ?\ eval(Exp),$ $v\_field2, v\_field3, ...,$ v_fieldn
$S' = S \backslash \{v\}$ where $v \in S,\ S \in \mathcal{L}(p,t)$ $, S' \in \mathcal{L}(t,p)$ $p \in P,\ t \in T$, v is a quantified variable containing n fields, $v[1] = Expression$ is a part of the precondition.	$type\_p \ ??\ eval(Exp),$ $v\_field2, v\_field3, ...,$ v_fieldn
$v' = Exp$ where $v' \in \mathcal{L}(t,p)$ $, p \in P,\ t \in T$.	$type\_p \ !\ Exp$
$S' = S \cup \{(Exp_1, Exp_2, ..., Exp_n)\}$ where $S \in \mathcal{L}(p,t),\ S' \in \mathcal{L}(t,p)$ $, p \in P,\ t \in T$.	$type\_p!Exp_1, Exp_2, .., Exp_n$

from place p is retrieved, according to the constraint $v[1] = Exp$ in the precondition. For a simple variable v' on an output arc, a message is sent to the channel obtained from place p. For a quantified variable $v \in S$, if $S' = S \backslash \{v\}$ is a part of the postcondition, a message is retrieved by searching throughout the channel obtained from place p, according to the constraint $v[1] = Exp$ in the precondition. Besides the cases above, we need to deal with $\cup\{(Exp_1, Exp_2, ..., Exp_n)\}$ in case $S' = S \backslash \{v\} \cup \{(Exp_1, Exp_2, ..., Exp_n)\}$is a part of the postcondition, by sending a message to the channel obtained from place p, using the values of $(Exp_1, Exp_2, ..., Exp_n)$. Table 4 gives the general mapping. After firing the transition, $t\_is\_enabled$ is set to false.

Table 5 gives the mapping for the postcondition in Formula 6, in which $m[1]$ is replaced with $msg2\_field2$ and $n[1]$ is replaced with $msg2\_field3$ as the precondition since we do not declare variables in PROMELA for the first field of each simple variable or quantified variable.

Line 38-47 in Appendix 4 is the resulting PROMELA code.

Table 5. Mapping from the postcondition in Formula 6 to PROMELA Expressions

Basic Relational Expression	PROMELA Code
$msg2[1] = transfer$	$type\_msg\_in?eval(transfer), msg2\_field2, msg2\_field3, msg2\_field4$
$A2' = A2\backslash\{m\}$	$type\_AbWorld??eval(msg2\_field2), m\_field2, m\_field3;$
$\backslash\{n\}$	$type\_AbWorld??eval(msg2\_field3), n\_field2, n\_field3;$
$\cup\{(m[1], (m[2] - msg2[4]), m[3])\}$	$type\_AbWorld!msg2\_field2, m\_field2 - msg2\_field4, m\_field3;$
$\cup\{(n[1], (n[2] + msg2[4]), n[3])\}$	$type\_AbWorld!msg2\_field3, n\_field2 + msg2\_field4, n\_field3;$

Step 4. Define an inline function for each transition. Each transition has its precondition and postcondition, we define an inline function for each transition $t \in T$ using the inline functions for its precondition and postcondition. Firing a transition is defined as atomic operations using the PROMELA keyword *atomic*.

```
inline t()
{
        is_enabled_t(); /*Set t_is_enabled to true/
           false*/
        if
        ::  t_is_enabled -> atomic{fire_t()}
        ::  else -> skip
        fi
}
```

For example, Line 48-54 in Appendix 4 is the inline function for transition *AbPurseTransfer* in Fig. 1.

Step 5. Define a process for the whole net. The dynamic semantics of a Petri net is to non-deterministically fire enabled transitions. We define the following PROMELA process with a loop to capture the dynamic semantics of a Petri net.

```
proctype ModelName(){
 bool t1_is_enabled = false;
 bool t2_is_enabled = false;  ...
 bool tn_is_enabled = false;
 do
    ::t1()
    ::t2()    ...
    ::tn()
  od
}
```

where $T = \{t_1, t_2, ...t_n\}$. For example, we define a process as Line 55-62 in Appendix 4, for the abstract model of Mondex in Fig. 1.

Step 6. Define the initial marking and run the processes. Let $P = \{p_1, ..., p_n\}$, for each place $p \in P$, with initial marking $M_0(p) = \{m_1, m_2, ..., m_k\}$. We define *type_p* ! m_i for each i, $i \in 1..k$ and run the process *`ModelName`*.

```
init {
    type_p1!m1;... type_p1!mk1;
    ...
    type_pn!m1;... type_pn!mkn;
    run ModelName()
}
```

For example, we define Line 63-67 in Appendix 4 for the abstract model of Mondex in Fig. 1, according to Formula 7.

3.3 Translation Correctness

[KG02] proposed a framework for translating models and specifications, in which atomicity of transitions and variables with unspecified next values were discussed as issues in translation. In our work, we use the *`atomic`* keyword in PROMELA to make the transition atomic, and we use temporal logic to specify the post-condition for each variable.

We introduce the definitions of completeness and consistency before defining translation correctness. Completeness ensures that each place, transition and initial marking has its representation in PROMELA code.

Definition 1. Translation Completeness: *Each entity in a Petri net is mapped to a language construct in* PROMELA.

Lemma 1. *Given a Petri net N, there exists a* PROMELA *program P_N representing N.*

Proof. The rules in Section 3.2 cover the translation from N to P_N.

Consistency ensures that the PROMELA code preserves the semantics of a Petri net. While there are several well known semantic models of Petri nets, we adopt the interleaving semantics, which is adequate for studying the system properties defined in temporal logic.

Definition 2. Translation Consistency: *The dynamic behaviour of a Petri net is preserved in* PROMELA *code. The interleaved execution is a sequence $\sigma =$*

$M_0t_0M_1t_1...t_{n-1}M_n$, where $n \geqslant 0$, $M_i(i \in \mathbb{N} \wedge 0 \leqslant i \leqslant n)$ is a marking and $t_i(i \in \mathbb{N} \wedge 0 \leqslant i < n)$ is a transition firing. PROMELA *code execution is $\sigma' = S_0Run(p_{t_0})S_1Run(p_{t_1})...Run(p_{t_{n-1}})S_n$, where $S_i(i \in \mathbb{N} \wedge 0 \leqslant i \leqslant n)$ is a snapshot of values in variables defined in* PROMELA *code, and $Run(p_{t_i})(i \in \mathbb{N} \wedge 0 \leqslant i < n)$ denotes the execution of inline function p_{t_i} translated from t_i following the rules in Section 3.2.*

Lemma 2. *(Initial Marking Consistency) The initial marking of a Petri net N is consistent with the initial values of variables in the translated* PROMELA *program P_N.*

Proof. According to Step 1 in Section 3.2, marked places are translated into channels, and Step 6 in Section 3.2, the initial marking is used to initialize the channel variables. The initial marking of a Petri net N is M_0, and S_0 is the snapshot of initial values of variables in the translated PROMELA program P_N. According to Step 6 in Section 3.2, S_0 is mapped from M_0.

Lemma 3. *(Semantic Consistency) P_N bisimulates N.*

Proof. Let σ be an execution of N, we proof P_N simulates N by induction on the length of sequence n.

Base case, $n = 0$. It is the initial marking consistency proved above.

Suppose it is true for $n = k$ that the claim holds, i.e., $\sigma = M_0t_0M_1t_1...t_{k-1}M_k$ is consistent with $\sigma' = S_0Run(p_{t_0})S_1Run(p_{t_1})...Run(p_{t_{k-1}})S_k$.

If $n = k + 1$, as the Step 2 in Section 3.2, the precondition of p_{t_k} is the mapping of precondition of t_k; as the Step 3 in Section 3.2, the postcondition of p_{t_k} is the mapping of postcondition of t_k, that is, S_{k+1} is the mapping of M_{k+1}; as the Step 4 in Section 3.2, $Run(p_{t_k})$ generates S_{k+1}, which denotes marking M_{k+1} obtained from firing t_k. So, $\sigma_{k+1} = M_0t_0M_1t_1...t_kM_{k+1}$ is consistent with $\sigma'_{k+1} = S_0Run(p_{t_0})S_1Run(p_{t_1})...Run(p_{t_k})S_{k+1}$.

The reverse direction is proved in the same way, hence, P_N bisimulates N.

Definition 3. Translation Correctness *consists of translation completeness and translation consistency.*

Theorem 1. *Given a Petri net N, the* PROMELA *program P_N obtained from the translation rules in Section 3.2 preserves the semantics of N.*

Proof. We prove the translation correctness by proving translation completeness and consistency. It is straightforward from Lemma 1 to 3.

3.4 Analysis Result

There are two security properties to verify for Mondex [SCW00], the details of these properties are listed in Table 6.

We use the model checker SPIN to verify the properties in exhaustive mode. Formula 9 and 10 are the LTL properties we used in SPIN to do verification, in which $bal\_sum = \sum_{a \in A, A \in AbWorld} a[2]$ is the sum of balances, $lost\_sum =$

Table 6. The Properties of Mondex to Verify

Property Name	Property Description
All Value Accounted	all value must be accounted for in the system: the sum of all purses' balances and lost components does not change.
No Value Created	no value may be created in the system: the sum of all the purses' balances does not increase.

$\sum_{a \in A, A \in AbWorld} a[3]$ is the sum of lost amounts, and 450 is exactly the sum of *bal_sum* and *lost_sum* in the initial marking.

$$\Box\, bal\_sum + lost\_sum = 450 \tag{9}$$

$$\Box\, bal\_sum \leqslant 450 \tag{10}$$

The verification result is that all these LTL properties are satisfied in the given initial marking.

4 Related Works and Discussion

Several research groups around the world have tackled this 1st pilot project in recent years. In [FW07], Z/Eves was used to mechanize the original specification of Mondex in Z [SCW00], which took about eight weeks to complete the mechanization of the entire specification, refinement and its proof. In [Ram07], Alloy was used to specify Mondex and the Alloy Analyzer was used to check the specification that resulted in the discovery of several bugs. The specification and analysis took about 6 months for a research internship to finish. [HSGR07] used the KIV to specify and verify Mondex using a single refinement, which took about one person month. [BY07] presented an Event-B specification of Mondex using B4free, which consists of 10 levels, an abstract model and 9 levels of refinement. The development took approximately 2 weeks of total effort spread over several months. In [GH07], RAISE was used to specify Mondex. The specification consists of three levels: abstract, intermediate, and concrete. Half of the proofs were done automatically.

Other works on Mondex mainly focus on the automation of the proof of Mondex, while [GH07] not only made effort on proof of Mondex, but also did some model checking with limits such that there are only 2 purses in the world, and money is in the range 0 to 3, to reduce states as much as possible. Our approach using model checking offers great scalability to verify the properties of Mondex.

Regarding the translation from Petri net to Promela, this paper offers a unique way to translate high level Petri net to Promela. [AGCH+08] provides an approach to translate Sam to Promela in which the embedded C code was used as the main approach, while we do not use embedded C code. [GG01] had the similar idea to ours on translation rules from Petri net to PROMELA, but it only dealt with low level Petri nets, while we propose an approach to translating high level Petri nets to PROMELA code.

We provide a way of using model checking to verify the formal specification of Mondex in SAM [ZLH08], including the abstract model and concrete model. This paper presents the abstract model as an example.

Effort

It took us two person months to complete the specification[ZLH08], and 80 person-hours to translate the SAM model into PROMELA code for Mondex concrete model and to verify the model automatically using SPIN.

Bugs Found

[Ram07] found three bugs in the Z specification, in which one bug is for missing constraints about authenticity, also found by KIV method [HSGR07], two bugs are related with reasoning errors during refinement. For the authenticity bug, the Z specification gives no constraints for authenticity so that a purse could be making a transaction with a non-authentic purse. For example, a purse is in *epv* status, which is *to* purse, waiting for *val* message, there should be constraints preventing this purse from receiving *req* message as *from* purse. Similarly there should also be constraints preventing the purse in *epa* status as *from* purse from receiving *val* message as *to* purse. Without these constraints for authenticity, the actual role of a purse could be inconsistent in the transaction. The other two bugs are both for reasoning errors during refinement which is not present in this paper as we use model checking and do not do that refinement. Our specification avoids the authenticity bug through adding proper constraints and does not have refinement bugs.

Scalability

We conducted the model checking of Mondex concrete model with a Windows based PC which has 1.8Ghz CPU and 2GB memory. Since the Mondex system is not a network system and only contains atomic operations involving two purses;

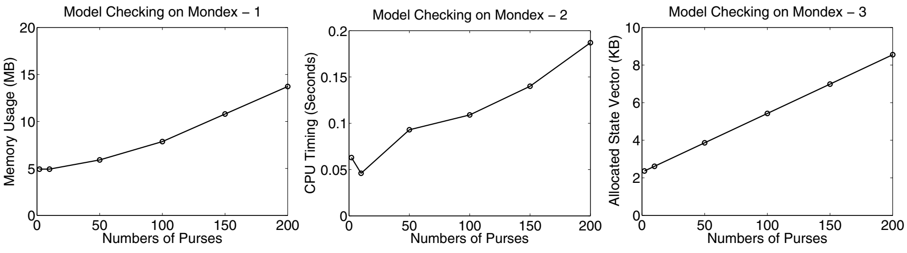


Fig. 2. Scalability of Model Checking on Mondex

it is adequate to model and analyze the system with one randomly chosen initial message. Therefore, we created a random message in the initial markings, the range for value of money was $0 \ldots 2^{31} - 1$. We conducted an experiment by increasing the number of purses in the initial markings, to show the scalability of memory usage, cpu timing and allocated state vector, as the Fig. 2 shown.

Acknowledgments. This work was partially supported by the NSF of U.S. under award HRD-0317692, HRD-0833093, IIP-0450482 and Presidential Fellowship of Florida International University. The authors would like to acknowledge Zhuo Sun for her contribution of ideas on the experimentation, and anonymous reviewers for their valuable comments.

References

[AGCH+08] Argote-Garcia, G., Clarke, P.J., He, X., Fu, Y., Shi, L.: A Formal Approach for Translating a SAM Architecture to PROMELA. In: SEKE, pp. 440–447 (2008)

[BY07] Butler, M., Yadav, D.: An incremental development of the Mondex system in Event-B. Form. Asp. Comput. 20(1), 61–77 (2007)

[CRC08] Grand Challenges in Computer Research, United Kingdom Computing Research Committee (2008), http://www.ukcrc.org.uk/grand_challenges/index.cfm

[FW07] Freitas, L., Woodcock, J.: Mechanising Mondex with Z/Eves. Form. Asp. Comput. 20(1), 117–139 (2007)

[GG01] Gannod, G.C., Gupta, S.: An Automated Tool for Analyzing Petri Nets Using SPIN. In: ASE 2001: Proceedings of the 16th IEEE international conference on Automated software engineering, Washington, DC, USA, 2001, p. 404. IEEE Computer Society, Los Alamitos (2001)

[GH07] George, C., Haxthausen, A.E.: Specification, proof, and model checking of the Mondex electronic purse using RAISE. Form. Asp. Comput. 20(1), 101–116 (2007)

[HD02] He, X., Deng, Y.: A Framework for Developing and Analyzing Software Architecture Specifications in SAM. The Computer Journal 45(1), 111–128 (2002)

[HM05] He, X., Murata, T.: High-Level Petri Nets - Extensions, Analysis, and Applications. In: The Electrical Engineering Handbook, Elsevier Academic Press, Amsterdam (2005)

[Hol03] Holzmann, G.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley Professional, Reading (2003)

[HSGR07] Haneberg, D., Schellhorn, G., Grandy, H., Reif, W.: Verification of Mondex electronic purses with KIV: from transactions to a security protocol. Form. Asp. Comput. 20(1), 41–59 (2007)

[KG02] Katz, S., Grumberg, O.: A Framework for Translating Models and Specifications. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. LNCS, vol. 2335, pp. 145–164. Springer, Heidelberg (2002)

[Ram07] Ramananandro, T.: Mondex, an electronic purse: specification and refinement checks with the Alloy model-finding method. Form. Asp. Comput. 20(1), 21–39 (2007)

[SCW00] Stepney, S., Cooper, D., Woodcock, J.: An Electronic Purse: Specification, Refinement, and Proof. Technical monograph PRG-126, Oxford University Computing Laboratory (July 2000)
[VSR07] Verified Software Repository (2007), http://vsr.sourceforge.net
[Woo06] Woodcock, J.: First Steps in the Verified Software Grand Challenge. Computer 39(10), 57–64 (2006)
[WSC$^+$07] Woodcock, J., Stepney, S., Cooper, D., Clark, J., Jacob, J.: The certification of the Mondex electronic purse to ITSEC Level E6. Form. Asp. Comput. 20(1), 5–19 (2007)
[ZLH08] Zeng, R., Liu, J., He, X.: A Formal Specification of Mondex Using SAM. In: SOSE 2008: Proceedings of the 2008 IEEE International Symposium on Service-Oriented System Engineering, Jhongli, Taiwan, pp. 97–102. IEEE Computer Society, Los Alamitos (2008)

Appendix: A Promela Program Translated from Abstract Model of Mondex

```
#define BOUND_msg_in 10
#define BOUND_AbWorld 10
#define BOUND_msg_out 10
chan type_AbWorld=[BOUND_AbWorld] of {short, int, int};
mtype = {aNullIn, transfer};
chan type_msg_in = [BOUND_msg_in] of {mtype, short,
    short, int};
mtype = {aNullOut};
chan type_msg_out = [BOUND_msg_out] of {mtype};
int bal_sum = 450,lost_sum = 0,seed = 0,last_seed = 0;
inline is_enabled_AbIgnore() {
    short msg1_field2;short msg1_field3;int msg1_field4;
    type_msg_in?<aNullIn,msg1_field2, msg1_field3,
        msg1_field4> ->
            AbIgnore_is_enabled = true
}
inline fire_AbIgnore(){
    type_msg_in?aNullIn,msg1_field2, msg1_field3,
        msg1_field4;
    AbIgnore_is_enabled = false
}
inline AbIgnore(){
    is_enabled_AbIgnore();
    if
    ::  AbIgnore_is_enabled -> atomic{fire_AbIgnore()}
    ::  else -> skip
    fi
}
inline is_enabled_AbPurseTransfer(){
    short msg2_field2, msg2_field3;int msg2_field4;
    int m_field2, m_field3, n_field2, n_field3;
    type_msg_in?<transfer,msg2_field2, msg2_field3, msg2_field4>;
    if
    ::  msg2_field2 != msg2_field3 &&
        type_AbWorld??[eval(msg2_field2), m_field2, m_field3] &&
        type_AbWorld??[eval(msg2_field3), n_field2, n_field3] ->
            AbPurseTransfer_is_enabled = true
    ::  else -> skip
    fi
}
inline fire_AbPurseTransfer() {
    type_msg_in?transfer,msg2_field2, msg2_field3, msg2_field4;
    type_AbWorld??eval(msg2_field2), m_field2, m_field3;
    type_AbWorld??eval(msg2_field3), n_field2, n_field3;
    atomic{type_AbWorld!msg2_field2, m_field2 - msg2_field4, m_field3;
    bal_sum = bal_sum - msg2_field4;}
    atomic{type_AbWorld!msg2_field3, n_field2 + msg2_field4, n_field3;
    bal_sum = bal_sum + msg2_field4;}
    AbPurseTransfer_is_enabled = false
}
inline AbPurseTransfer() {
    is_enabled_AbPurseTransfer();
    if
    ::  AbPurseTransfer_is_enabled -> atomic{ fire_AbPurseTransfer()}
    ::  else -> skip
    fi
}
proctype AbstractMondex (){
    bool AbIgnore_is_enabled = false;
    bool AbPurseTransfer_is_enabled = false;
    do
    :: AbIgnore ()
    :: AbPurseTransfer ()
    od
}
init {
    type_msg_in!transfer,1,2,50;type_AbWorld!1,100,0;
    type_AbWorld!2,200,0; type_AbWorld!3,150,0;
    run AbstractMondex()
}
```

Formal Modelling of Separation Kernel Components

Andrius Velykis and Leo Freitas

University of York, UK
andrius@velykis.lt, leo@cs.york.ac.uk

Abstract. Separation kernels are key components in embedded applications. Their small size and widespread use in high-integrity environments make them good targets for formal modelling and verification. We summarise results from the mechanisation of a separation kernel scheduler using the **Z/Eves** theorem prover. We concentrate on key data structures to model scheduler operations. The results are part of an experiment in a Grand Challenge in software verification, as part of a pilot project in verified OS kernels. The project aims at creating a mechanised formal model of kernel components that gets refined to code. This provides a set of reusable components, proof strategies, and general lemmas. Important findings about properties and requirements are also discussed.

Keywords: Kernel, grand challenge, formal models, proof.

1 Introduction

Although software is ubiquitous, it is still perceived as an "Achilles' heel" of most systems, often being a serious threat. There is increasing evidence of the successful use of formal methods for software development. In [22], 62 industrial projects over 20 years are discussed. The survey explains the effect that formal methods have on time, cost, and quality of systems and how their application is becoming cost effective, hence easier to justify, not as an academic pursuit or legal requirement, but as a business case. By the use of mathematical analysis, formal methods enable accurate definition of a problem domain with capability of proving properties of interest. Formal methods application usually produces reliable evidence for errors that are fiendishly difficult to catch. Industrial and academic researchers have joined up in an international Grand Challenge (GC) in Verified Software [21], with the creation of a Verified Software Repository (VSR) with two principal aims: (i) construction of verified software components; and (ii) industrial-scale verification experiments to drive future research in the development of theory and tool support [2].

This paper is part of a pilot project within the GC in modelling OS kernels. It summarises work done in [19]. The project follows work on modelling smartcards [12] and flash memory file stores [8]. Our objective is to provide proofs of the correctness of a formal specification and design of kernels for real-time embedded systems. We start from Craig's formal models of OS kernels [6], and take into account separation kernel requirements set out by Rushby [15] and in the US National Security Agency's *Separation Kernel Protection Profile* [18].

We focus on formalisation of data structures needed by an abstract specification of a separation kernel scheduler, its main operations and algorithms. Our long experience with the Z notation [20] and one of its theorem provers (**Z/Eves** [16]) is well aligned with Craig's original model [6] also developed in Z, as well as existing resources in the VSR. Kernel development work is facilitated by reusing modelling concepts and proof tactics of mechanising simple kernel components (*e.g.,* basic types and the process table) from [10]. A complex separation kernel process table adds process separation, external identifiers and other structures to address architecture and security requirements. While the process queue is shared with minor differences between both kernels, the shedulers are architecturally different. The increased complexity and structural differences affect associated proofs — although some lemmas can be reused, in general the proofs are different in both kernels and thus both contribute different verified components to VSR. With mechanisation and formal modelling we upgrade Craig's original separation kernel model from [6] by: improving the specification adding missing invariants and new security properties; verifying API robustness and model correctness in general; *etc.* Note that all details of results presented in this paper, analysis, justification as well as formal specification and proof scripts, are available in [19].

We briefly set the kernel verification scene in the next section. Section 3 presents a case study, which involves mechanisation of key data structures for the kernel's scheduler: a process table that keeps track of user and device process information; a process queue used by scheduler operations and algorithms; the scheduler invariant itself; and proved properties of interest. In Section 4, we reflect on our results by giving some measures. Finally, Section 5 sums up our findings and sets the agenda for future work.

2 Background

Craig's book on kernels [6] includes Z specifications and refinement of simple and separation kernels developed as an exercise that is beyond academic. It serves as a starting point for our project. The objectives are to demonstrate feasibility of top-down development using formal specification and verification with refinement to code (*i.e.,* correctness by construction) [11]. Craig's original models are typeset by hand and include several manual proofs. We augment the specification that uses Z notation [20] by mechanising it with a theorem prover [16] in order to more precisely record its correctness arguments from hand-written proofs. Given Craig's expertise as a kernel developer, we try to keep as faithful as possible to his original designs, only changing it at places where identified mistakes have been made. All results [19], including models, lemmas, *etc.* are being curated in the VSR [2] at SourceForge (`vsr.sourceforge.net`).

Verification of OS Kernels. An OS kernel is key in coordinating all access to underlying hardware resources like processors, memory, and I/O devices. Application processes can access these resources via system calls and inter-process communication. Kernel development has a reputation for being a complex task for two prime reasons: (i) every system requires the kernel to provide correct functionality and good performance; and (ii) the kernel cannot make (direct) use of the abstractions it provides (*e.g.,* processes, semaphores, *etc.*).

Microkernels for embedded systems are a suitable target for formal verification due to their small size and controlled environment. Such verification is an industrial-scale exercise that is undertaken in a number of academic and commercial projects. We identify two different approaches to verification. One starts with an existing kernel (possibly code or concrete design) and verifies its properties bottom-up, *e.g.,* Microsoft's hypervisor — a separation kernel aiming at virtualisation of hardware [5]. The project has verified existing C and assembler code for the functional correctness of kernel memory models. Within the GC, there is a recently started pilot project on the verification of FreeRTOS open-source kernel [3] that involves scientists in the UK and India. Alternatively, a top-down approach starts with the formalisation of high-level requirements that then gets refined (as formally as possible) to code. This approach allows reasoning about kernel properties without being bound to an existing implementation. Its application can be seen in parts of the commercial project L4.verified [13], which formalises and verifies a high-performance general-purpose microkernel; and in the work on Xenon [14], a security hypervisor based on the Xen OS. The former uses Isabelle/HOL to specify, abstract and verify properties of a Haskell prototype of the kernel, whereas the latter is using Z and CSP to model the C code. With properties proved about such formal model, one can then apply refinement techniques to obtain concrete designs. Furthermore, abstract components facilitate development of new kernel structures, where their properties are proved without an implementation. Our project aims to create fully generic abstract kernel models and refine them to code with good levels of automation — this paper contains results from the beginning of this large scale effort.

Simple kernel components based on [6] have already been mechanised in [10,9]. There we found interesting issues, including missing and hidden invariants. Although Craig's models have great insight from an OS engineer in necessary underlying data types, a series of mistakes are introduced, both clerical and more substantial in design decisions. Craig's work also includes a C implementation for Intel's IA32 architecture that is carried out using data refinement and the Z refinement calculus [20]. This paper continues with separation kernel components, in particular the scheduler, as reported in [19].

Separation Kernel. Separation kernel architecture was first introduced by Rushby [15], where different kinds of processes are isolated to achieve desirable security properties. The US National Security Agency has produced a Separation Kernel Protection Profile (SKPP) under the Common Criteria certification framework to define requirements for separation kernels used in environments that require high-robustness [18]. SKPP also includes the kernels' interaction with both hardware and firmware platforms, hence these components also need to be verified. In here, we assume them as *trusted entities* verified elsewhere. In our work, we follow the formal models by Craig [6], which are relatively close to the SKPP requirements, as extensively discussed in [19]. Craig assumes the kernel to be running on an *Intel's IA32/64* platform, and verbally states that memory partitioning and context switches are achieved by the underlying hardware. We need to specify this mathematically in order to support statements spanning the kernel and the hardware. The main concerns are to ensure separation of process address spaces: they must execute in isolated memory partitions; inter-process communication is only allowed via vetted communication channels; and so on. To achieve this, one needs to have memory partitioning in the kernel, where each process is allocated a dedicated area. Process communication is established via message passing over a special shared memory area. Unauthorised communication between processes is prevented by having external process identifiers, which are translated into internal representations within the kernel. The requirement to have process execution separation is achieved by a non-preemptive scheduler. It ensures only a single component is active at each given time. Craig models device processes as trusted code running within the kernel. Our work here is to mechanise, polish, and improve Craig's original model. Also, we know from collaboration that the modelling of the Xenon hypervisor [14], which is a much more complex kernel, is benefiting from ideas presented here and in [19].

3 Case Study

Separation kernel formal model development is a significant undertaking due to the high number of different components and requirements, as well as specific domain knowledge involved. This case study presents some details on mechanisation, modelling and verification of a separation kernel specification, based on hand-written models by Craig [6]. The mechanisation has four stages: (i) parsing and typechecking Z for syntactic type consistency; (ii) domain and axiomatic checking that shows well-formedness of expressions (*e.g.,* functions are applied within their domains), and soundness of axioms; (iii) feasibility lemmas providing an existence proof for the initial state, and operation preconditions showing API robustness and correct state invariant; and finally (iv) proving conjectures that represent properties of the model. We also summarise key data structures in the scheduler like a process table and scheduling queue. This paper focuses on formal modelling, hence details of the mechanical proof process are omitted here. Instead, all theorems, proofs and complete analysis are available in [19]. A very detailed report on proving Z specifications with **Z/Eves** is provided in [7].

3.1 Process Table

A core data structure in our separation kernel is a *process table* (*PTab*) that stores all process information. Previous work on simple kernels [10] has been reused, with additional variables and invariants to address process separation security requirements. All kernel processes are referenced by their identifiers, as bounded non-empty range type of *PID*s. Also, to distinguish between user processes and "trusted" platform code (*i.e.,* device processes), we use Z free types ($PTYPE ::= uproc \mid dproc$), which in this case are enumerated type constructors that form a partition. This means that *uproc* and *dproc* are distinct, and the only elements of the set *PTYPE*. Outside the kernel, processes are referenced by external identifiers to prevent unauthorised access to internal kernel resources. Craig [6, Chap. 5] suggests having an unbounded number of external user process identifiers *UPID*, and a limited number of device identifiers *Dev*, since in embedded environments device configuration is known from the start.

PTab

$$used, free : \mathbb{F}\ PID;\ nup : UPID;\ dmap : Dev \rightarrowtail\!\!\!\!\!\!+\ PID;\ pidext : PID \mathrel{\ooalign{$\rightarrow$\cr\hidewidth$+$\hidewidth}} UPID$$
$$ptype : PID \nrightarrow PTYPE;\ extpid : UPID \nrightarrow PID;\ state : PID \nrightarrow PSTATE$$

$$free = PID \setminus used \land used = \operatorname{dom} state = \operatorname{dom} ptype \land pidext = extpid^{\sim}$$
$$\exists\, dprocs, uprocs : \mathbb{F}\ PID \bullet dprocs = ptype^{\sim}(\!|\ \{\, dproc \,\}\ |\!) = \operatorname{ran} dmap$$
$$\land\ uprocs = ptype^{\sim}(\!|\ \{\, uproc \,\}\ |\!) = \operatorname{ran} extpid$$
$$\forall\, u : UPID \mid u \geq nup \bullet u \notin \operatorname{dom} extpid$$

A process table (*PTab*) is specified using a Z *schema*: a labelled record data structure with invariants. *PTab* is similar to the one for a simple kernel in [10]: it has finite sets ($\mathbb{F}$) *used* and *free* for process identifiers (*PID*) that are disjoint; and it specifies partial function ($\nrightarrow$) mappings for each *used PID* to access various related process information. Functions *ptype* and *state* specify type and process-state information for each process, respectively. Available process states are defined by the free-type *PSTATE* like *PTYPE* above, and omitted here. Process table invariants require these mappings to exist for all *used* processes (*i.e.,* functions recording process information are total on *used*).

External identifiers are different for each process type and stored in separate structures. Device numbers are stored in *dmap*: a partial injective ($\rightarrowtail\!\!\!\!\!+$) relationship, which guarantees a one-to-one mapping between device numbers *Dev* and kernel device process *PID*s. For user process identifiers, we have functions *extpid* and *pidext* as the inverse ($^{\sim}$) of each other to allow simple bi-directional identifier queries. Some ambiguity while modelling a system in Z is common practice, providing it aids clarity and simplify proof goals. Because *dmap*, *extpid* — and a few other process information mappings not included above — are used for processes of different types, just like with *used PID*s for *state* and *ptype* domains, we need to identify device and user-process sets. Since these sets are images of *ptype* for each process type, we define sets *dprocs* and *uprocs* locally using an existential quantifier for device and user processes, respectively. These sets are linked with the range of their corresponding functions — as well as the domain of the few mappings not shown here. Finally, *nup* defines next available *UPID* for user processes. Proving precondition of *PTab* operations revealed the necessity for ensuring a future unused *UPID* ($u \notin \operatorname{dom} extpid$) for all *UPID*s beyond (and including) *nup*, hence the last invariant. More details on how this appeared are given below, and a full account is given in [19, Chap. 6].

Properties. *PTab* specifies an injective relationship between device process external (*Dev*) and internal (*PID*) identifiers. A corresponding property of user process identifiers (*UPID*) can be formulated as

$$PTab \vdash extpid \in UPID \rightarrowtail\!\!\!\!\!\!\!\!\nrightarrow PID \land pidext \in PID \rightarrowtail\!\!\!\!\!\!\!\!\nrightarrow UPID \tag{1}$$

Given a *PTab* state (*i.e.,* there exists an instance of the state where its invariant holds: we have a feasible model), both *extpid* and *pidext* are injective ($\rightarrowtail\!\!\!\!\!\!\!\!\nrightarrow$), thus for each internal user process *PID*, there exists a unique external user process *UPID*, and vice-versa. We proved it as a theorem in **Z/Eves** using current *PTab*'s invariants. This way we achieve separation of concerns, as the theorem can be used later in proofs. As a redundant invariant in *PTab*, it could unnecessarily complicate future proofs, *e.g.,* extra proofs are needed per redundant property. We formalised various operations for process identifier management, such as process allocation and deletion.

Process Table Refinement. Mechanisation involves refinement of data structures to accommodate proofs and model changes. A detailed case study of *PTab* refinement is given in [19, Chap. 6] and describes the evolution of the original schema in [6, p. 220] to the final form as shown above. Refinement relationships are proved in [19] for each step to ensure that the original specification is still satisfied, *e.g.,* $PTab \Rightarrow PTabv4 \Leftrightarrow PTabv3 \Leftrightarrow \ldots$. Equivalence ($\Leftrightarrow$) changes represent specification refactoring without changing the original meaning, whereas a refinement $Spec \sqsubseteq Design$, here as reverse implication $Design \Rightarrow Spec$, guarantees that the stronger schema satisfies all properties of the one it refines, and is used to correct mistakes and add missing invariants. Some examples illustrating different refinement steps are given below (details and all proofs are in [19]).

Complicated mathematical constructs can be replaced with more suitable Z idioms for theorem proving without compromising corresponding *PTab* schema. Originally the set of device processes was defined as set comprehension (middle)

$$dprocs = \{\, p : PID \mid p \in used \land ptype(p) = dproc \,\} = ptype^{\sim}(\!|\, \{\, dproc \,\} \,|\!) \tag{2}$$

which is straightforward enough to understand: *dprocs* is a set of known (*used*) *PID*s with *dproc* type, as required. However, set comprehension expressions are difficult to reason about in proofs, as they require pointwise extensionality proofs (*i.e.,* to show that every element in the set satisfies its invariants). Instead, if we could characterise the same relationship with higher-level operators, we would then be likely to have higher automation levels. So, (2, middle) can be refactored as the relational image of the of inverse of *ptype* (2, right). Function *ptype* maps *PID*s to their process types (*PTYPE*). Inverting it, we get a relation (set of pairs) between *PTYPE* and their corresponding *PID*s, which might no longer be functional as more than one identifier might be of user or device type. Then, we apply relational image ($R(\!|\ S\ |\!)$) over this resulting set of pairs, which gives a set of values in *ptype* for all its *dproc*-typed members. Thus, (2, right) gives all *PID*s of *dproc* type in *ptype* as set *dprocs*. The advantage of this equivalent formulation is that it can take advantage of a series of lemmas about inverse and relational image from the Z mathematical toolkit [17], which then leads to more automatic proofs. The equality in (2) enables us to prove equivalence between the schemas affected by this change (*i.e.,* $PTabv4 \Leftrightarrow PTabv3$ in [19, p. 46]).

Failed precondition proofs during the mechanisation of a deterministic *UPID* allocation algorithm by Craig [6, p. 222] revealed a missing invariant. The monotonic increment of new *UPID*s did not ensure that new identifiers are unused by the process table. We refine the *PTabv*4 schema (*i.e.,* $PTab \Rightarrow PTabv4$) by adding an invariant (3) requiring the next and subsequent *UPID*s to be available.

$$\forall\, u : UPID \mid u \geq nup \bullet u \notin \mathrm{dom}\ extpid \tag{3}$$

Without it, one could not prove that *UPID* allocation operations kept the after state of *extpid* values being injective. This is an expected, yet missing invariant both in the Z models and in the English-written requirements, rather than a cosmetic model change. We benefit from streamlined specification, corrected invariants and clarity of requirements in the refined *PTab* schema, while the proofs show schema conformity to the original specification. The full account on *PTab* refinement is given in [19, Chap. 6].

3.2 Process Queue

The separation kernel scheduler stores waiting processes in *process queues* to ensure correct execution order. We model them with operations to access and manage queue elements, such as querying for elements, enqueue and dequeue processes to be scheduled, and so on. The resources being queued are process identifiers (*PID*s) from the underlying process table (*PTab*). During these queue operations *PTab* components are kept read-only.

The next (horizontal) schema *PQ* defines the queue as an injective sequence of process identifiers: it is a function from ordered pairs of indexes started from one to unique *PID* values (*i.e.,* akin to $\mathbb{N}_1 \rightarrowtail\!\!\!\!\rightarrow PID$). Only elements from the given *PTab*'s *used* set are allowed in the queue, as indicated by the subset ($\subseteq$) constraint over the sequence's range of *PID*s queued. In Z, schema inclusion like *PTab* is used to factor complex data structures into its constituent components.

$$PQ\ \widehat{=}\ [\,PTab;\ pr : \mathrm{iseq}\ PID \mid \mathrm{ran}\ pr \subseteq used\,] \quad PQInit\ \widehat{=}\ [\,PQ' \mid pr' = \langle\rangle\,]$$

The queue is initialised as empty in *PQInit*. In Z, dashed variables like pr' represent operation's after state, where PQ' represents dashed components from *PQ*. Note that in *PQInit* the underlying *PTab* is not restricted during *PQ* initialisation, hence allowing *PTab* to be initialised before (*e.g.,* with a suitable *PTabInit*). Such need arises when *PQInit* is used in scheduler initialisation (see Sect. 3.3). Because we require only an existence proof for the feasibility of such initial state, this arrangement is just right (*e.g.,* $\exists\, PQ' \bullet PQInit$). We need two separate process queues for user processes and devices. To avoid name clash, a *DQ* schema is defined by renaming the queue sequence variable *pr* to *dv* as

$$DQ \ \widehat{=} \ PQ[dv/pr] \qquad DQInit \ \widehat{=} \ PQInit[dv'/pr']$$

Notice that the underlying *PTab* for both *PQ* and *DQ* remains the same. This neat alignment/reuse of components is a great feature of the Z schema calculus [20, Chap. 11]. Craig suggests an injective sequence [6, Sect. 3.4] in *PQ* to ensure that no process can be queued multiple times. Injective sequence updates are tricky because they require uniqueness of sequence range elements, yet sequence operators (*e.g.,* concatenation $\_ \frown \_$) are not aware of such restrictions. This requires additional effort during precondition proofs to guarantee *PID* queueing uniqueness. For instance, to concatenate an element x to a sequence t one just needs to say $t' = t \frown \langle x \rangle$, whereas for a injective sequence s, it is also necessary to show that x is unique in s (*i.e.,* $x \notin \text{ran } s$) like

$$\forall s : \text{iseq } X; \ x : X \bullet x \notin \text{ran } s \Rightarrow s \frown \langle x \rangle \in \text{iseq } X \tag{4}$$

which is proved in [10]. Given a non-empty generic type X and an injective sequence s, if an element $x \in X$ is not mapped in s, then appending x to s keeps the result injective. Lemmas like this have been reused across various pilot projects, and form the basis of a general library, which is one of the outcomes of the Grand Challenge: reusable components and proofs. Furthermore, process queue is a generic kernel data structure and is used within other components (*e.g.,* semaphores in simple kernel [6, Sect. 3.7]). Different kernel process tables mandate redoing *PQ* proofs, however, proof structures and toolkit lemmas are successfully reused, hence minimising implementation effort. Further reuse is facilitated by collecting such verified components in the VSR.

3.3 Scheduler

Like in [15], our separation kernel employs a round-robin scheduler. It is non-preemptive, hence a running process can only suspend voluntarily or terminate. Such approach is chosen for its simplicity and current use in embedded systems. Scheduler includes operations for making a process *ready*, *suspended*, or to *terminate* processes. Synchronous device *I/O* in the kernel is achieved by the scheduler executing device processes at a higher priority than user processes, so that when a device call is performed, a corresponding device process is executed to handle the call, while the user process suspends and waits for a reply.

Schema *Sched* contains two ready-queues for each process type: *PQ* and *DQ*. Two special (distinct) identifiers are used for the idle (*ip*) and current (*cr*) processes. The former is used when no other processes are scheduled, whereas the latter stores the process currently executing. We also define a component to aggregate all queued process identifiers in a single set (*queued*). This auxiliary term allows us to write simpler expressions, entailing easier proofs later.

$$\begin{array}{|l}
\textit{Sched} \\ \hline
PQ;\ DQ;\ cr, ip : PID;\ queued : \mathbb{P}\ PID \\ \hline
queued = \operatorname{ran} pr \cup \operatorname{ran} dv \land \{\ cr, ip\ \} \subseteq used \land cr \notin queued \land ip \notin queued \\
\operatorname{ran} pr \subseteq ptype^{\sim} (\!|\ \{\ uproc\ \}\ |\!) \land \operatorname{ran} dv \subseteq ptype^{\sim} (\!|\ \{\ dproc\ \}\ |\!)
\end{array}$$

The state invariant has that the current and idle processes are known to the kernel process table (*i.e.,* within *used*), and must not be queued for execution — no process can be executing and waiting at the same time. We ensure that processes of different types are queued accordingly: *pr* for (*uproc*) user processes, and *dv* for (*dproc*) kernel device processes. We use subset containment over the relational image of each function.This specification of the scheduler is a substantial upgrade from Craig's [6, Sect. 5.6]. The addition of a reference to *PTab* via *PQ* enabled the specification of process properties. Thus, we were able to convert informal requirements given in English in the book: that *PQ* is for user processes and *DQ* is for devices, for instance. Also with the invariants on *cr*, *ip* being known (in *used*) *PID*s only, we could specify and prove kernel security properties, and define robust operations for the scheduler (*i.e.,* operations that are proved to account for all behaviours as successful and exceptional cases).

We initialise the scheduler with empty queues and an idle process running that is passed as ($p?$) an input variable. In Z, inputs are tagged with a question mark. We keep the modular approach for kernel components and reuse previously defined operations: during scheduler initialisation, queues and process table are initialised by corresponding operations like *PQInit*.

$$SchedInit \mathrel{\widehat=} [\,Sched';\ PQInit;\ DQInit;\ p? : PID \mid ip' = p? \land cr' = ip'\,]$$

$$SchedPTabInit \mathrel{\widehat=} PTabInit \mathbin{⨾} (AddIdleProcess \land SchedInit[ip!/p?]) \setminus (ip!, u!)$$

Using *PTab* operations defined elsewhere [19, Chap. 6], we can construct full initialisation of *PTab* and *Sched* as *SchedPTabInit*. We use schema composition for the sequential execution of operations. In Z, the schema composition ($S \mathbin{⨾} T$) operator uses the after state of S as the before state of T, with the after state of T being the overall after state, and similarly for the before state of S. That works well providing the after state components of S is the whole of the before state of T (*i.e.,* we have homogeneous composition). This models a process table that is initialised first by *PTabInit*, then an idle process is allocated by *AddIdleProcess* (*i.e.,* a user process allocation operation in [19, p. 57]), and passed into scheduler initialisation *SchedInit*. Finally, output variables $ip!$ and $u!$ are hidden (*i.e.,* existentially quantified) to avoid exposing them outside the operation. It neatly reuses definitions plumbed with the schema calculus, which are equivalent to the following expanded schema.

$$\begin{array}{l}
SchedPTabInitExpand \mathrel{\widehat=} [\,Sched' \mid used' = \{\ ip'\ \} \land nup' = 2 \land cr' = ip' \land \\
\ pr' = dv' = dmap' = \emptyset \land extpid' = \{\ 1 \mapsto ip'\ \} \land ptype' = \{\ ip' \mapsto uproc\ \} \land ..\,]
\end{array}$$

It shows state initialisation after all updates take place. The idle process ip' is allocated with expected initial values and passed into the scheduler for execution.

3.4 Scheduler Operations

Abstract data type operations in Z are specified as a relation between before and after states. This includes both successful and exceptional cases. Typically this is given as a disjunction of schemas. Usually we negate the successful case precondition, where each error case accounts for some of the negated predicates, such that the overall precondition equates to *true*, hence leading to a robust interface. This is the so-called Oxford style of Z [20]. A common and simple solution for error handling is to report the error whilst keeping the state constant. Our separation kernel model uses an *errors with memory* [1, Sect. 18.3] approach to store the error message in between operations. This way a subsequent operation can check whether the kernel is in a valid state. Furthermore, Craig [6] defines error cases to kill the kernel via an interrupt as a simplistic, albeit blunt an approach for secure exit. Nevertheless, this interrupt handling is not modelled formally. To aid this we need to extend it by defining and proving kernel error handling security properties. Note that an interrupt-like error handling approach, with state validation, recovery and logging, has been successfully modelled in Mondex project [12]. Detailed description of kernel error handling is given in [19, Chap. 5], while here we present a short summary of key operations only.

For example, a process queueing operation *EnqPQOk* performs the sequence concatenation (queueing) in the successful case. The error cases, such as when the process is already queued (*ErrQueued*), are defined separately and then disjoined with the successful case. In Z, ΔPQ indicates both PQ and PQ' are included, yet without any constrains over state variables. ΞPQ is just like ΔPQ but requires everything in PQ to be kept constant.

$$\begin{array}{lcl} \mathit{EnqPQOk} & \widehat{=} & [\,\Delta PQ;\ \Xi PTab;\ p? : PID \mid pr' = pr \frown \langle p? \rangle\,] \\ \mathit{ErrQueued} & \widehat{=} & [\,\Xi PQ;\ \Delta ErrV;\ p? : PID \mid p? \in \operatorname{ran} pr \land serr' = errqueued\,] \\ \mathit{EnqPQ} & \widehat{=} & (\mathit{EnqPQOk} \land \mathit{SysOk}) \lor \mathit{ErrQueued} \lor \ldots \end{array}$$

A process is added for scheduling by "readying" it. This means it must be enqueued and its state in *PTab* must be set to *psready*. Such queueing must keep everything else constant (*i.e.,* queueing happens before scheduling). We reuse *EnqPQ* operation to perform the actual queueing in *EnqSchedOk*.

$$\begin{array}{lcl} \mathit{EnqSchedOk} & \widehat{=} & [\,\Delta Sched;\ EnqPQ \mid cr' = cr \land ip' = ip \land dv' = dv\,] \\ \mathit{EnqUserSched} & \widehat{=} & \mathit{EnqSchedOk} \lor \mathit{ErrNotUserPID} \lor \ldots \end{array}$$

Schema *EnqUserSched* models queueing a user process without updating the current, idle or device processes. The operation performs a number of checks to ensure that all invariants are satisfied. Included schema *EnqPQ* has a robust specification for queue-level queueing with: successful cases; case when a process is unknown to the kernel (*e.g., PID* not in *used*); or case when it is already queued (*e.g., PID* in *pr* range). Since no queue operation alters *PTab* ($\Xi PTab$), *EnqPQ* ensures that the underlying process table does not change as a result of *EnqSchedOk*. Scheduler invariants give rise to additional error cases: a process can have the wrong type; it might already be running; the idle process cannot be queued; and so on. These error cases are defined to create a total enqueue operation *EnqUserSched*. The complete "readying" operation *MakeReady* enqueues and sets process state. We use schema composition ($\fatsemi$) to update the *pstate* function in *PTab* for a given process ($p? \in PID$) to *psready* in *SetReady*.

$$MakeReady \mathrel{\widehat=} EnqUserSched \fatsemi ((IsSysOk \land SetReady) \lor ErrKeepFail)$$

The input variable comes from *EnqPQ*. Note that process state changes only if the enqueue succeeds, hence we check for expected system state with *IsSysOk*. If *EnqUserSched* does fail (*i.e.,* either via *EnqPQ* errors, or by its own error cases), the error is propagated by *ErrKeepFail*. An analogous operation has been defined for device processes. In [4], these schemas are used as APIs for the definition of a parallel/distributed scheduling algorithm that considers the behavioural aspects of the specification, rather than what is happening in the data structures.

The main scheduler function is to determine and execute processes in an appropriate sequence. As mentioned, device processes have priority over user processes, while the idle process is run when nothing else is scheduled. We model this via separate operations, later on conjoined to represent the overall scheduling algorithm. All operations follow the same pattern: a process is selected for execution and its state is set via a *PTab* operation (*SetRunning*).

$$\begin{array}{|l}
\multicolumn{1}{l}{RunIdleNext} \\
\hline
\Delta Sched;\ SetRunning[ip/p?];\ SysOk \\
\hline
dv = pr = \langle\rangle \land cr' = ip \land ip' = ip \land pr' = pr \land dv' = dv \\
\hline
\end{array}$$

Since the idle process is never queued in the scheduler, running it does not require updating scheduler queues, hence we can specify everything with operation *RunIdleNext*. Like *SetReady*, *SetRunning* updates *pstate* for a given $p?$ input to *psrunning*, which in this case is the idle process *ip*. This operation can only be executed when both process queues *dv* and *pr* are empty. The operation sets the idle process as current ($cr' = ip$), sets it to running, and does not change anything else. Finally, *SysOk* signals that operation has been successful. User and device scheduling operations are similar in that they both take the first element in a respective queue and change its state to running. For dequeue, corresponding operations for device and user processes queues are used.

$$\begin{aligned}
SchedUserNext &\mathrel{\widehat=} [\,\Delta Sched;\ DequeuePQ \mid dv = \langle\rangle \land pr \neq \langle\rangle \land \\
&\qquad\qquad cr' = p! \land ip' = ip \land dv' = dv\,] \\
RunUserNext &\mathrel{\widehat=} (SchedUserNext[n/p!] \fatsemi SetRunning[n/p?]) \setminus (n)
\end{aligned}$$

Schema *SchedUserNext* sets process $p!$, which is output by *DequeuePQ* operation, as the current process. Note that *DequeuePQ* is responsible for updating user process queue pr'. The scheduling algorithm is specified in the operation invariants: output user process $p!$ is scheduled when device queue *dv* is empty, and user process queue *pr* is not. *RunUserNext* appends the *SetRunning* operation to the scheduling algorithm. Here auxiliary variable n is used to link the output parameter from one operation to the input parameter of the next. We can formulate and prove statements about scheduling operation properties to improve assurance in them, *e.g.,* we show that *RunUserNext* always executes the first element in user process queue (*i.e.,* $RunUserNext \vdash cr' = head\ pr$). The device scheduling operations *SchedDeviceNext* and *RunDeviceNext* are specified in an similar manner with *DequeueDQ* and with the precondition that the device queue is not empty ($dv \neq \langle\rangle$). The complete API for any scheduler state implements the scheduling algorithm by disjoining all top-level operations.

$$SchedNext \mathrel{\widehat{=}} RunIdleNext \lor RunUserNext \lor RunDeviceNext$$

We proved that the precondition of this operation is *true*, meaning that it can be executed for any scheduler state: it is a robust operation that will always succeed. Even more, we proved a conjecture to show that the operation never fails: it will always return the system call was okay (*e.g.,* $SchedNext \vdash serr' = sysok$). The composing operation invariants ensure that the total operation *SchedNext* is never in a state where queueing error case preconditions may apply. Obviously, all that did not fall into place neatly. It was the result of a well-crafted model, followed by the ruthless scrutiny of the theorem prover, alongside appropriate guidance in the process via useful lemmas for acceptable levels of automation.

With the definitions of enqueue and scheduling available, we can create a suspend operation. A process is never preempted, but can suspend voluntarily to relinquish execution to other processes. One of the cases when this may happen is during inter-process communication: a process sends a message to another process and suspends itself to allow the other process to eventually execute and handle / reply the message. The suspend operation runs the next available process and places the caller back in the process queue. This is specified by reusing *SchedNext* and *MakeReady* operations.

$$RequeueUserProcess \mathrel{\widehat{=}} (SchedNext \mathbin{⨾} MakeReady)$$

Both *SchedNext* and *MakeReady* are robust (*i.e.,* precondition is *true*), and *SchedNext* constrains all variables in *Sched* referenced in *MakeReady* (*i.e.,* we have a homogeneous operation), hence we can safely combine them using schema composition. That is, since we have already shown *SchedNext* will always succeed, it is not necessary to check if the system is in a valid state before executing *MakeReady*. Nevertheless, as *MakeReady* can fail due to a queueing error, the full operation is not atomic: we could not recover the scheduler's state in case *MakeReady* fails. The need for atomicity of such operations is discussed in [4].

The scheduler reuses a large number of operations from *PTab* and *PQ*. The nested schemas and multiple error cases create complex operations that can be compactly presented using the schema calculus. Yet, expanding all definitions could lead to rather tricky proof steps, and a good amount of lemmas need to be in place if one is to complete the proofs with acceptable levels of automation. Fortunately, as it often happens [10,9], such lemmas are not only repeated, but also reusable from previous experiments, which minimises the overall effort.

4 Discussion

General outcomes of the separation kernel component mechanisation confirm the findings of the previous projects [10] — proper tool support and a verification framework build confidence in the formal specification. Syntax errors are eliminated, model feasibility and API robustness are verified, and missing invariants to guarantee correct operations are found. The formal model is fully proved mechanically — the proofs help establishing the correctness of the Z specification. The validity of the kernel model must be demonstrated by proving architectural and security properties. Our work significantly improves Craig's scheduler model. By translating verbal requirements to mathematical invariants and improving design of the specification, we are able to formulate and prove certain properties about the components (*e.g.,* the scheduler deadlock analysis below). The main separation kernel properties of process separation (*e.g.,* memory partitioning and communication via established channels only), however, span a number of kernel components, some of which have not been mechanised yet. Thus these properties must be analysed and proved as a future exercise.

Scheduler Deadlock Analysis. Using invariants in scheduler schema *Sched*, we prove that kernel starvation by queueing all known processes, hence none would be available for execution, is impossible: *i.e.,* $\forall Sched \bullet queued \subset used$ (Sect. 3.3). Nevertheless, a deadlock can occur, when, for example, the initial process (which creates other processes) suspends without "readying" other processes. In this case, the idle process would be running with all processes queued. Even so, this functionality is specific to the process, not the kernel, and we do not formally model the running processes themselves.

Other Separation Kernel Components. The process table, queue and scheduler are core data structures within a separation kernel. The next step is to mechanise and model other components, such as messaging or memory management, to achieve and verify full separation of processes. We have laid the foundations for that: external process identifiers can be allocated and translated in the process table to avoid exposure of kernel internal representation; process schedule and suspend functionalities are the core of the inter-process messaging subsystem. The formal model of underlying hardware platform would go beyond Craig's original model, but would allow proving memory access restrictions. Craig assumes hardware exceptions [6, p. 204], yet does not formally specify them. A formal model for interrupts would benefit from similar experience with Mondex smartcards [12].

Proofs & Benchmarks. In [19] we have already mechanised close to five separation kernel specification sections (out of 12 [6, Chap. 5]). The remaining part constitutes of memory management and messaging components, as well as kernel interface spanning all components. Our mechanisation of three core components have a total of 263 paragraphs comprising schemas, types, and axioms, with ~40% of these being related to operation feasibility proofs (*i.e.,* operation preconditions). These generated 254 verification conditions: ~50% are about model feasibility; 35% are about proof automation rules in **Z/Eves**; and ~10% are related to model refactoring. Also, we proved 12 properties of interest (~5%), several of which have been presented in the case study (Sect. 3). Furthermore, the general theories created by other pilot projects contain well over 120 reusable theorems about various mathematical data types [9]. Separation kernel proofs were discharged with over 1300 proof commands, 23% of which require creative steps involving quantifier's witnesses, or knowledge on how the theorem prover works. The remaining 77% involved proof exploration steps of moderate difficulty and straightforward/blind tasks, given the right model. These numbers enable comparison with previous GC pilot projects, which accumulated similar information. The Z mechanisation of Mondex [12, p. 117–139] has 25% less paragraphs, yet has over 350% more proof: that is largely due to the underlying refinement calculation involved. Various automation lemmas from Mondex were reused. This improved our share of push-button proof steps to 40%, whereas they were about 27% in Mondex. This suggests that Mondex proofs were more difficult, yet the separation kernel proofs automation levels benefited from that work. Such difference is expected as we did not do any refinement proofs yet: it will be worthwhile comparing those later. The work presented here took approximately 900 man-hours, which also included the first author mastering the **Z/Eves** prover without previous experience, and writing up the thesis in [19].

5 Conclusions

The Grand Challenge's pilot projects inspire us to model and verify various application domains. One aim beyond actual mechanisation is to make it easier for the next team who want to work with kernels. For that, we provide data types and useful lemmas that are central to modelling kernel scheduling. Basic verified data structures were developed for a simple kernel [10], with a collection of general lemmas from the Verified Software Repository (VSR) being reused [9].

In this paper we improve the specification and formal model of the separation kernel in [6]. Separation kernel data structures address security requirements and are more complex than their simple kernel counterparts. Most of the work consisted of identifying properties about data types, calculating preconditions for (*i.e.,* feasibility of) each operation, and verifying everything via formal proof along the way. This mechanisation revised and improved the specification of the process scheduler and its associated components: it corrected modelling errors on data types, as well as missing error cases of operations, and mistaken invariants from [6], some of which were discussed here (and discussed in full in [19]). Together with new extracted general lemmas and detailed verification process report (all available in [19]), we believe this to be an important contribution in building theories for mechanised formal modelling of OS kernels.

Future Work. We aim to complete the formal kernel model and prove the process separation, as well as to perform formal refinement of kernel components to a concrete model and implementation code. We intend to explore how such kernel model is used in other projects [14], and different scheduling algorithms [4].

Acknowledgements. We are grateful to Iain Craig for his useful account of formal kernel specification and modelling. The first author is now employed by the EPSRC (EP/H024050/1) AI4FM project at Newcastle University, UK.

References

1. Barden, R., et al.: Z in Practice. Prentice Hall, Englewood Cliffs (1994)
2. Bicarregui, J., et al.: The verified software repository. Formal Aspects of Computing 18(2), 143–151 (2006)
3. Berry, R.: A free real-time operating system (FreeRTOS)
4. Boerger, E.: Refinement of distributed agents. In: Dagstuhl Seminar 09381 (2009)
5. Cohen, E., et al.: VCC: A practical system for verifying concurrent C. In: Urban, C. (ed.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
6. Craig, I.D.: Formal Refinement for Operating System Kernels. Springer, Heidelberg (2007)
7. Freitas, L.: Proving Theorems with Z/Eves. T. Report, University of Kent (2004)
8. Freitas, L., et al.: Posix and the verification grand challenge: A roadmap. In: 13th ICECCS, pp. 153–162. IEEE Computer Society, Los Alamitos (2008)
9. Freitas, L.: Extended Z mathematical toolkit – Verified Software Repository. Technical Report CRG13, University of York (2008)
10. Freitas, L.: Mechanising data-types for kernel design in Z. In: Oliveira, M.V.M., Woodcock, J. (eds.) SBMF 2009. LNCS, vol. 5902, pp. 186–203. Springer, Heidelberg (2009)
11. Hall, A., Chapman, R.: Correctness by Construction: Developing a Commercial Secure System. IEEE Software 19(1), 18–25 (2002)
12. Jones, C., Woodcock, J. (eds.): Formal Aspects of Computing: Special Issue on the Mondex Verification, vol. 20(1). Springer, Heidelberg (2008)
13. Klein, G., et al.: seL4: Formal verification of an OS kernel. In: 22nd ACM Symposium on Operating Systems Principles (SOSP). ACM, New York (2009)
14. McDermott, J., Freitas, L.: Formal security policy of Xenon. In: FMSE (2008)
15. Rushby, J.M.: Design and verification of secure systems. ACM SIGOPS Operating Systems Review 15(5), 12–21 (1981)
16. Saaltink, M.: Z/Eves 2.2 User's Guide. Technical report, ORA (1999)
17. Saaltink, M.: Z/Eves 2.2 Mathematical Toolkit. Technical report, ORA (2003)
18. SKPP: U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness, v.1.0.3. National Security Agency (June 2007)
19. Velykis, A.: Formal modelling of separation kernels. Master's thesis, Department of Computer Science, University of York (2009)
20. Woodcock, J., Davies, J.: Using Z. Prentice-Hall, Englewood Cliffs (1996)
21. Woodcock, J.: First steps in the verified software grand challenge. IEEE Computer 39(10), 57–64 (2006)
22. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal Methods: Practice and Experience. ACM Computing Surveys (2009) (in Press)

Mechanized Verification with Sharing

Gregory Malecha and Greg Morrisett

Harvard University SEAS

Abstract. We consider software verification of imperative programs by theorem proving in higher-order separation logic. Separation logic is quite effective for reasoning about tree-like data structures, but less so for data structures with more complex sharing patterns. This problem is compounded by some higher-order patterns, such as stateful iterators and visitors, where multiple clients need to share references into a data structure. We show how both styles of sharing can be achieved without sacrificing abstraction using mechanized reasoning about fractional permissions in Hoare Type Theory.

1 Motivation

Axiomatic semantics [7] is one way to formally reason about programs. Under these semantics, programs are analyzed by considering the effect of primitive operations as transformers of predicates over the state of the system. Unfortunately, stating and reasoning about these predicates is complicated in the presence of shared, mutable pointers. It was not until Reynolds proposed separation logic [16] that reasoning about pointer-based imperative programs in a modular way became tractable. However, even with this logic some verification tasks are far from simple, particularly when we cannot easily describe which abstractions "own" pointers, due to sharing.

The difficulty comes from conflicting goals: We want to reason locally and compositionally about programs, and, at the same time, we wish to share data to make algorithm and data structure implementations more efficient. Vanilla separation logic provides the first, but makes the second difficult because of non-local effects as illustrated by the following Java program:

```
1 void error(List<T> lst) {
2    Iterator<T> itr = lst.iterator ();
3    lst .remove(0);
4    itr .next(); // throws ConcurrentModificationException
5 }
```

This method creates an iterator object referencing the beginning of a list. Successive calls to the `next()` method are intended to step through the list. However, the call on line 3 removes the element that the iterator references, destroying the iterator's view on the list, even though line 3 does not mention the iterator. If problems like this go undetected at run-time, they can result in

A. Cavalcanti et al. (Eds.): ICTAC 2010, LNCS 6255, pp. 245–259, 2010.

`NullPointerExceptions` in Java, or memory corruption or segmentation faults in lower level languages such as C.

Separation logic can be used to reason about such programs, because it incorporates a notion of ownership: a command is allowed to access memory only if it appears in its "footprint" (i.e., pre- or post-condition). This gives us a frame property that ensures properties on sub-heaps that are *disjoint* from the footprint of the command are preserved, giving us an effective way to reason about abstracted, compound commands (i.e., method calls). However, in the example above, there is no way to give the iterator ownership over the list and still make the direct modification to the list at line 3. Rather, the iterator abstraction and the client must share the list. Consequently, after the state modification on line 3, we can no longer be ensured that the iterator's internal invariants hold, invalidating the call to `next()`.

Unfortunately, as originally conceived, separation logic is too restrictive to validate many patterns where an iterator or visitor safely shares state with the client or other abstractions. For example, if we called `lst.length()` on line 3, then the internal invariants of the iterator remain preserved, and we can safely call the `next()` method. But traditional separation logic does not support this kind of reasoning without exposing implementation details of the abstraction.

In this paper we show how type-directed formal verification can be used to verify data structures that share state, in particular collections and their iterators. Our data structures are heap-allocated and make liberal use of pointer aliasing. We have found that sharing makes formally reasoning about the correctness of programs in an automated way difficult, and we believe general theorem proving techniques are most suitable to address these problems.

We consider sharing of two sorts, *external* and *internal*. In *external* sharing, we wish to support multiple, simultaneous views of the same underlying memory, as in the iterator example above. In *internal* sharing, we wish to completely hide the sharing from the client so he/she can reason with a simple interface while the implementation is free to use aliasing for correctness or efficiency.

Contributions

We begin with a brief overview of the Ynot verification library [4] (Section 2), which provides an effective tool for writing and reasoning about higher-order, imperative programs with a type system that incorporates a form of separation-style, Hoare logic. We then:

- Show how fractional permissions [8] can be applied to provide sharing of high-level abstractions, focusing on collections. (Section 3)
- Show how external sharing can be leveraged to mechanically verify higher-order, effectful computations in Ynot, focusing on iterators. (Section 4)
- Show how internal sharing can be expressed by describing the representation of B+ trees, and how our approach simplifies the implementation of an iterator for traversing the leaves of the tree. (Section 5)
- While formalizing B+ trees, we also show a technique for formalizing data structures with a non-functional connection to their specification. (Section 5)

In our presentation, we focus on interfaces in stylized Coq, but our implementation and verification are available at `http://ynot.cs.harvard.edu/`. After our contributions, we consider the burden of verification, the implications of our techniques, and related work (Section 6).

We believe that our methodology extends previous work describing aliasing in separation logic [3] by being amenable to machine-checkable proofs and embeddable in Hoare-type theory. Previous work has developed paper-and-pencil proofs and, as has been seen in other contexts [1], the evolution from rigorous, manual proofs to mechanically verified proofs is not always straightforward.

2 Background

Ynot [4] is a Coq library that implements Hoare type theory [14] to reason about imperative programs using types. Hoare logic describes commands using Hoare triples, commands along with pre- and post-conditions. Ynot encodes these in the type of the `Cmd` monad.

$$\{P\}\, c\, \{r \Rightarrow Q\} \quad \equiv \quad c \,:\, \texttt{Cmd}\,(P)\,(r \Rightarrow Q)$$

where the command c has pre-condition P and post-condition Q that depends on the return value of c (bound to r). This type means that c can be run in any state that satisfies P and, if c terminates with value r, then the resulting state will satisfy Q.

Ynot defines pre- and post-conditions in the logic of Coq as predicates over heaps, which, themselves, are defined as functions from pointers to optional values. Previous work [4] showed how using a stylized fragment of separation logic makes verification conditions more amenable to automation and therefore less burdensome for the programmer to prove. As in previous work, we use a shallow embedding of separation logic which we extend with support for fractional permissions (Figure 1).

The empty heap (`emp`) denotes a heap containing no allocated cells (i.e., all pointers are mapped to `None`[1]). The permission to access the heap cell pointed to by p is given by the fractional points-to relation, $p \overset{q}{\mapsto} v$ [8,6,15]. We use the simple model of fractional permissions originally developed by Boyland [8]. In this work, the value of q is a rational number such that $0 < q \leq 1$. In all cases the points-to relation asserts that the heap contains a cell with the value v addressed by the pointer p. When $q = 1$, the points-to assertion grants code the ability to read, write, and deallocate the cell. When $q < 1$, the points-to relation gives read-only access to the heap cell. Informally, the separating conjunction ($*$) states that the two conjuncts hold on two "disjoint" pieces of the heap (denoted $h_0 \perp h_1$.) In the presence of fractional permissions, two heaps are disjoint if each pointer is mapped by only one heap or the values are the same and the fractions sum to a valid fraction. The $\uplus$ operator defines the merger of disjoint heaps. The

[1] The symbols `None` and `Some` are the constructors of the `option` α type which represents an optional value of type α which is included in the `Some` constructor.

$\boxed{h \models P}$ Heap Propositions (*hprop*)

Empty	$h \models \mathtt{emp}$	$\overset{\Delta}{\Longleftrightarrow}$	$\forall p.\ h\,p = \mathtt{None}$
Points-to	$h \models p \overset{q}{\mapsto} v$	$\overset{\Delta}{\Longleftrightarrow}$	$h\,p = \mathtt{Some}\,(q, v) \wedge \forall p'.\ p \neq p' \to h\,p = \mathtt{None}$
Separate Conjunction	$h \models P * Q$	$\overset{\Delta}{\Longleftrightarrow}$	$\exists h_1\,h_2.\ P\,h_1 \wedge Q\,h_2 \wedge h = h_1 \uplus h_2 \wedge h_1 \perp h_2$
Existentials	$h \models \exists x. P_x$	$\overset{\Delta}{\Longleftrightarrow}$	$\exists x.\ P_x\,h$
Pure Injection	$h \models [p]$	$\overset{\Delta}{\Longleftrightarrow}$	$\mathtt{emp}\,h \wedge p$

$\boxed{h_0 \perp h_1}$ Heap Disjointness

$$h_0 \perp h_1 = \forall p. \begin{cases} v_0 = v_1 \wedge q_0 + q_1 \leq 1 & \forall i \in \{0,1\}.\ h_i\,p = \mathtt{Some}(q_i, v_i) \\ q_i \leq 1 & i \in \{0,1\} \wedge h_i\,p = \mathtt{Some}(q_i, v) \wedge h_{1-i}\,p = \mathtt{None} \\ \mathtt{True} & \forall i \in \{0,1\}.\ h_i\,p = \mathtt{None} \end{cases}$$

$\boxed{h_0 \uplus h_1}$ Heap Union

$$(h_0 \uplus h_1)\,p = \begin{cases} \mathtt{Some}(q_0 + q_1, v) \leq 1 & \forall i \in \{0,1\}.\ h_i\,p = \mathtt{Some}(q_i, v) \\ \mathtt{Some}(q_i, v) & i \in \{0,1\} \wedge h_i\,p = \mathtt{Some}(q_i, v) \wedge h_{1-i}\,p = \mathtt{None} \\ \mathtt{None} & \forall i \in \{0,1\}.\ h_i\,p = \mathtt{None} \end{cases}$$

Fig. 1. The shallow embedding of separation logic used in Ynot

`new`	$: \Pi(T : \mathtt{Type})(v : T), \mathtt{Cmd}(emp)(p : ptr \Rightarrow p \mapsto v)$
`free`	$: \Pi(p : ptr), \mathtt{Cmd}(\exists T, \exists v : T, p \mapsto v)(\_ : unit \Rightarrow emp)$
`read`	$: \Pi(T : \mathtt{Type})(p : ptr)(P : T \to hprop),$ $\mathtt{Cmd}(\exists v : T, p \overset{q}{\mapsto} v * P\ v)(v : T \Rightarrow p \overset{q}{\mapsto} v * P\ v)$
`write`	$: \Pi(T : \mathtt{Type})(p : ptr)(v : T), \mathtt{Cmd}(\exists T, \exists v' : T, p \mapsto v')(\_ : unit \Rightarrow p \mapsto v)$
`bind`	$: \Pi(T\,U : \mathtt{Type})(P P' : hprop)(Q : T \to hprop)(Q : U \to hprop),$ $(\forall v : T, Q\,v \Longrightarrow P') \to \mathtt{Cmd}(P)(Q) \to (T \to \mathtt{Cmd}(P')(Q')) \to \mathtt{Cmd}(P)(Q')$
`return`	$: \Pi(T : \mathtt{Type})(v : T), \mathtt{Cmd}(\mathtt{emp})(r : T \Rightarrow [r = v])$
`cast`	$: \Pi(T : \mathtt{Type})(P\,P' : hprop)(Q\,Q' : T \to hprop), (P' \Longrightarrow P) \to$ $(\forall v : T.Q\ v \Longrightarrow Q'\ v) \to \mathtt{Cmd}(P)(v : T \Rightarrow Q\ v) \to \mathtt{Cmd}(P')(v : T.Q'\ v)$
`frame`	$: \Pi(T : \mathtt{Type})(P\,Q\,R : hprop), \mathtt{Cmd}(P)(r : T)(Q\,r) \to$ $\mathtt{Cmd}(P * R)(r : T \Rightarrow Q\,r * R)$

Fig. 2. Axiomatic basis for Hoare type theory using separation logic

separation logic of Ynot also supports existential quantification, and injection of pure propositions (propositions that do not mention the heap such as $n < 5$).

Ynot axiomatizes the primitive heap operations using the commands given in Figure 2. The `new` command allocates memory by producing the read-write capability to access the memory cell pointed to by the return value. The precondition specifies that the command needs no heap capabilities, and by the frame property, the resulting pointer must be fresh (disjoint from the rest of the heap). The `free` command deallocates a memory cell by consuming the read-write permission to access the cell. The `read` command reads the values from a cell given a predicate, P, that describes the rest of the heap based on this value. The dependence on P allows us to enforce that the v in the pre-condition is the same as the v in the post-condition because P could include a precise equation on v. For example, if p pointed to a pointer to v, we could pick P = **fun** $r \Rightarrow r \mapsto v$ thus making the post-condition reduce to $p \overset{q}{\mapsto} r * r \mapsto v$ which maintains the connection between p and v. The `write` command updates the value in a heap cell given a pointer and the new value.

These commands are combined using monadic `bind` and `return` in addition to a cast command that takes a proof and applies Hoare's consequence rule. The`frame` command extends the footprint of a command with extra capabilities that are invariant under the command. This is essential to local reasoning and enables Ynot to run a command with pre-condition P and post-condition Q in an environment satisfying $P * R$ and allows us to infer the post-condition $Q * R$.

3 Sharable Abstractions: Linked Lists

In this section, we develop the foundations of our contributions by defining a simple interface for externally sharable list structures. Sharing will allow multiple read-only views of the list or a single read-write view. We will achieve this using fractional permissions in the same way that we do for heap cells.

In Ynot, abstract data types are defined by a representation predicate and associated theorems and imperative commands. The interface for sharable lists (`ImpList`) is given as a type-class [18] in Figure 3. The class is parametrized by the type of the elements in the list (`T`) and the type of handles to the list (`tlst`). The representation predicate (llist) relates a fractional permission (of type perm), the list handle, and a functional model of the list (the list T) to the imperative representation, i.e. the structure of the heap described as a predicate over heaps (*hprop*). The heap proposition `llist q t l` states that `t` is a handle to a q-fraction of an imperative representation of the functional list `l`. Conceptually, we can think of this as $t \overset{q}{\mapsto} l$. Assuming this, `new` and `free` are analogous to Ynot's `new` and `free` commands. The specifications for `sub` and `insert` are expressed by relating their return value and post-condition to the result of pure functions (`specNth` and `specInsert`) that we take as specifications (we give the `specNth` function as an example of our specifications). We use the `#` in types to denote computationally irrelevant variables [4]. These can be thought of as compile-time-only values that are used to specify the behavior of computations without incurring run-time overhead.

One easy way to realize this interface is using singly-linked lists as shown in Figure 4. The following recursive equations specify the representation invariant for singly-linked list segments between pointers *from* and *to*.

```
Fixpoint specNth {T} (ls : list T) (n : nat) : option T :=
  match ls, n with
    | nil , _        ⇒ None
    | a :: _, 0    ⇒ Some a
    | _ :: b, S n ⇒ specNth b n
  end
Class ImpList (T : Type) (tlst : Type) := {
  (* tlst is a handle to perm capabilities to the list T *)
   llist  : perm → tlst → list T → hprop ;
  (* Fractional merging and splitting of lists *)
   llist_split  : ∀ q q' t m, q |#| q' →
     llist (q + q') t m ⟺ llist q t m ∗ llist q' t m ;
  (* Allocate an empty list *)
  new : Cmd (emp) (res : tlst ⇒ llist 1 res nil) ;
  (* Free the list *)
  free : Π (t : tlst),
    Cmd (∃ ls : list T, llist 1 t ls) (_ : unit ⇒ emp) ;
  (* Get the ith element from the list if it exists . *)
  sub : Π (t : tlst T) (i : nat) (m : #list T#) (q : #perm#),
    Cmd (llist q t m)
         (res : option T ⇒ llist q t m ∗ [res = specNth m i]);
  (* Insert an element at the ith position in the list . *)
  insert : Π (t : tlst) (v : T) (i : nat) (m : #list T#),
    Cmd (llist 1 t m)
         (_ : unit ⇒ llist 1 t (specInsert v i m))
}
```

Fig. 3. Externally-sharable list interface

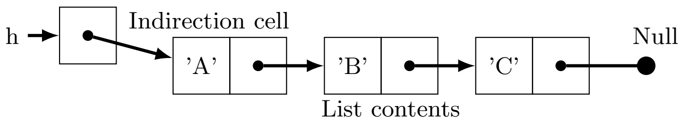


Fig. 4. A heap representing the list `['A', 'B', 'C']`

$$\text{llseg}\ q\ \mathit{from}\ \mathit{to}\ \texttt{nil} \overset{\Delta}{\Longleftrightarrow} [\mathit{from} = \mathit{to}] \tag{1}$$

$$\text{llseg}\ q\ (\text{Ptr}\ \mathit{from})\ \mathit{to}\ (a :: b) \overset{\Delta}{\Longleftrightarrow} \exists x.\ \mathit{from} \overset{q}{\mapsto} \text{mkNode}\ a\ x * \text{llseg}\ q\ x\ \mathit{to}\ b \tag{2}$$

$$\text{tlst}\ T \ = \ \text{ptr} \tag{3}$$

$$\text{llist}\ q\ t\ \mathit{ls} \overset{\Delta}{\Longleftrightarrow} \exists \mathit{hd}.\ t \overset{q}{\mapsto} \mathit{hd} * \text{llseg}\ \mathit{hd}\ \text{Null}\ \mathit{ls} \tag{4}$$

In equation (1), the model list is empty so the start and end pointers are the same. When the model list is not empty, i.e. it is a cons $(a :: b)$, *from* must not be null, and there must exist a pointer x such that *from* points to a heap cell containing a and x ($\mathit{from} \overset{q}{\mapsto} \text{mkNode}\ a\ x$) and x points to the rest of the list (llseg $q\ x\ to\ b$). Equations (3) and (4) make the list mutable by making `tlst` an indirection pointer so the pointer to the head of the list can change.

Since the definition only claims a q-fraction of the list, all of the points-to assertions have fraction q. This allows us to prove the llist_split lemma that states a $q + q'$ fraction of the list is equivalent to a q fraction of the list disjoint from a q' fraction of the list. We can use this proof to create two disjoint, read-only views of the same list to share.

4 External Sharing: Iterators

The ability to share the list abstraction pays off when we need to develop another view of the list. Here, we develop a simple and efficient iterator.

We define the iterator with a representation predicate (liter) and commands for creating (open), advancing (next), and deallocating (close) it:

```
Parameter titr : Type → Type.
Parameter liter : ∀ T. perm → tlst T →
  titr T → list T → nat → hprop.
Parameter open : Π (T : Type) (t : tlst T) (m : #list T#)
  (q : #perm#),
  Cmd (llist q t m) (res : titr T ⇒ liter q t res m 0).
Parameter next : Π (T : Type) (t : titr T) (m : #list T#)
  (idx : #nat#) (own : #tlst T#) (q : #perm#),
  Cmd (liter q own t m idx)
      (res : option T ⇒ liter q own t m (idx + 1) *
          [res = specNth m idx]).
Parameter close : Π (T : Type) (t : titr T) (own : #tlst T#)
  (m : #list T#) (q : #perm#),
  Cmd (∃ idx, liter q own t m idx) (_ : unit ⇒ llist q own m).
```

The heap proposition liter $q\ l\ i\ m\ n$ describes the iterator i to the n^{th} element of the imperative list l which is a representation of the functional list m. Here, the fractional permission q is the ownership of the underlying list, not of the iterator, so even if it is not 1, we will be able to modify the iterator, just not the underlying list. The open computation constructs an iterator to the beginning of a `tlst T` by converting the heap predicate from `llist q t m` to `liter q t res m 0`. The next command returns the current element in the list (or `None` if the iterator is past the end of the list) and advances the position, reflected in the index argument of liter . The close command reverses the effect of open by converting the liter back into a llist .

The own parameter specifies the owner of the iterator in the style of ownership types [5]. We make it a parameter so that we can specify the invariant precisely enough to prove that the post-condition to close is exactly the same as the pre-condition to open so that use of an iterator loses no information about the underlying list.

In this interface, an iterator requires full ownership of a heap cell containing a pointer to the current node, and fractional ownership of the owner pointer and underlying list. For simplicity, we break the specification of the list into two parts: the part that has already been visited (firstn $i\ m$) that goes from st to cur, and the rest (skipn $i\ m$) that goes from cur to Null.

$$\begin{array}{rcl}\text{titr}\,T & = & \text{ptr}\\ \text{iter}\ own\ q\ t\ m\ i & \overset{\Delta}{\Longleftrightarrow} & \exists cur.\exists st.\ t \mapsto cur\ *\ own \overset{q}{\mapsto} st\ *\\ & & \text{llseg}\ q\ st\ cur\ (\text{firstn}\ i\ m) * \text{llseg}\ q\ cur\ \text{Null}\ (\text{skipn}\ i\ m)\end{array}$$

5 Internal Sharing and Non-functional Heaps: B+ Trees

We now turn to the problem of internal sharing. Recall that in internal sharing, we completely hide the sharing from the client. To demonstrate our technique, we discuss the representation of B+ trees that we presented in previous work [12]. We choose B+ trees to implement this interface because they have a structure that is tricky to reason about because of aliasing and previous work only demonstrated an imperative `fold` rather than the more primitive iterator. Our implementation for this interface does not include fractional permissions, though we believe that it would be relatively straightforward to add them.

Figure 5 gives our target interface for finite maps and their iterators, which we combine for brevity. The class is parametrized by the type of keys, values, and finite map handles. The logical model is a sorted association list (`fmap K V`) that we relate to the handle with the heap proposition `repMap q t m`. The remaining computations are similar to those of the list; we support allocation, deallocation, key lookup, and key-value insertion. The iterator interface is the same as the list iterator interface except it does not have fractional permissions.

B+ trees are balanced, ordered, n-ary trees that store data only at the leaves and maintain a pointer list in the fringe to make in-order iteration of the values efficient. Figure 6 shows a simple B+ tree with arity 4. As with most tree structures, B+ trees are comprised of two types of nodes:

– Leaf nodes store data as a sequence of at most n key-value pairs in increasing order by key. The trailing pointer position points to the next leaf node.
– Branch nodes contain a sequence of at most n key-subtree pairs and a final subtree. The pairs are ordered such that the keys in a subtree are less than or equal to the associated key (represented in the figure as `treeSorted min max`). For example, the second subtree can only contain values greater than 2 and less than or equal to 6. The final subtree covers the span greater than the last key; in the figure, this is the span greater than 6.

As with the iterator, the two main difficulties in formalizing B+ trees reveal themselves in the representation predicate. The first is that the model does not totally determine the heap structure. The second is the, potentially non-local, pointer aliasing in the leaves.

```
(* tfmap is the type of finite maps from key to value. *)
Class FiniteMap (K V : Type) (tfmap : Type) := {
  (* The tfmap handle represents the fmap. *)
  repMap : tfmap → fmap K V → hprop ;
  new : Cmd emp (h : tfmap ⇒ repMap h nil) ;
  insert  : Π (h : tfmap) (k : K) (v : V) (m : #fmap K V#),
    Cmd (repMap t m)
        (res  : option V ⇒ repMap h (specInsert v m) *
            [res  = specLookup k m]) ;
  (** ... free & lookup ... **)
  (** The iterator **)
  titr  : Type ;
  repIter  : tfmap → titr → fmap K V → nat → hprop ;
  open  : Π (h : tfmap) (m : #fmap K V#),
    Cmd (repMap h m) (res : titr ⇒ repIter h res m 0) ;
  next : Π (h : titr) (own : #tfmap#) (m : #fmap K V#)
    (idx  : #nat#),
    Cmd (repIter own h m idx)
        (res  : option (K * V) ⇒
            repIter own h m (idx + 1) * [specNth m idx]) ;
  close  : Π (h : titr) (own : #tfmap#) (m : #fmap K V#),
    Cmd (∃i. repIter own h m i) (_ : unit ⇒ repMap own m)
}
```

Fig. 5. The imperative finite map interface

The standard way to address the first problem is to use a direct relational specification of the heap, existentially quantifying the splitting of the list into subtrees at each level [17]. While this works well for paper-and-pencil proofs, it makes automation difficult because we must witness these existentials in every place that we deconstruct the tree. To avoid this, we factor the relation between the interface model and the heap description into a relation and a function, as shown in Figure 7.

Our representation model is a functional tree that we index by the height to enforce the balancedness constraint. In Coq, we could define this as follows:

```
Fixpoint ptree (h : nat) : Type :=
  match h with
    | 0     ⇒ list (key * value)
    | S h' ⇒ list (key * ptree h') * ptree h'
  end
```

The second difficulty deals more directly with sharing. In the standard representation for a tree, we existentially quantify the pointers at the parent pointer for each node. However, when the rightmost leaf of one tree must alias the leftmost leaf of the next the pointers have disjoint scopes making it impossible to

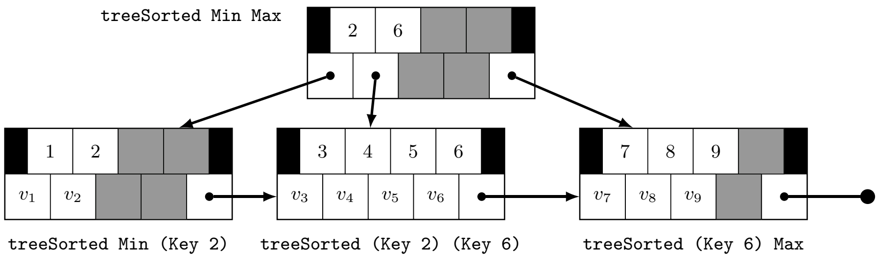


Fig. 6. A B+ tree of arity 4 ($n = 4$) for the finite map from $i \mapsto v_i$ for $1 \leq i \leq 9$

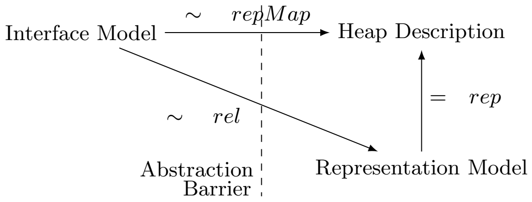


Fig. 7. Decouple the relational mapping between the interface and the heap by factoring out a representation model that is functionally related to the heap

relate them directly. Changing the representation to quantify the pointers at the lowest ancestor of the two nodes complicates the recursion because we have to handle the first subtree specially. This strategy also leads to difficulties when describing iterators because we will want to ignore the "trunk" and consider only the leaves. Instead, we embed the pointers directly in the representation model using the following type:

```
Fixpoint ptree (h : nat) : Type :=
  ptr * match h with
          | 0    ⇒ list (key * value)
          | S h' ⇒ list (key * ptree h') * ptree h'
        end
```

Using this representation model, we can easily describe the aliasing without worrying about scoping since all of the pointers are quantified at the root.

With this model, we can turn to defining the heap representation predicate. We define repTree h o p to hold on a heap representing the ptree p of height h when the rightmost leaf's next pointer equals o:

$$\texttt{repTree}\,0\;optr\;(p', ls) \overset{\Delta}{\Longleftrightarrow} \exists ary.\; p' \mapsto \texttt{mkNode}\,0\;ary\;optr\; * \;\texttt{repLeaf}\;ary\;ls$$

$$\begin{aligned}&\texttt{repTree}\;(1+h)\;optr\;(p', (ls, nxt)) \overset{\Delta}{\Longleftrightarrow}\\ &\quad \exists ary.\; p' \mapsto \texttt{mkNode}\;(h+1)\;ary\;(\texttt{ptrFor}\;nxt)\,*\\ &\qquad \texttt{repBranch}\;ary\;(\texttt{firstPtr}\;nxt)\;ls\; * \;\texttt{repTree}\;h\;optr\;nxt\end{aligned}$$

The repTree predicate has two cases depending on the height. In the leaf case, the array holds the list of key-value pairs from the ptree.

$$\begin{aligned}&\texttt{repLeaf}\ ary\ [v_1,\ldots,v_m] \overset{\Delta}{\Longleftrightarrow}\\ &\quad ary[0] \mapsto \texttt{Some}\ v_1 * \cdots * ary[m-1] \mapsto \texttt{Some}\ v_m *\\ &\quad ary[m] \mapsto \texttt{None} * \cdots * ary[n-1] \mapsto \texttt{None}\end{aligned}$$

In the branch case, the array holds key-pointer pairs such that each pointer points to the corresponding subtree. This is captured by the repBranch predicate. Note that we use ptrFor to *compute* the pointers from the model.

$$\begin{aligned}&\texttt{repBranch}\ ary\ optr\ [(k_1,t_1),\ldots,(k_m,t_m)] \overset{\Delta}{\Longleftrightarrow}\\ &\quad ary[0] \mapsto \texttt{Some}\ (k_1, \texttt{ptrFor}\ t_1) * \texttt{repTree}\ h\ (\texttt{firstPtr}\ t_2)\ t_1\ * \cdots *\\ &\quad ary[m-1] \mapsto \texttt{Some}\ (k_m, \texttt{ptrFor}\ t_m) * \texttt{repTree}\ h\ optr\ t_m\ *\\ &\quad ary[m] \mapsto \texttt{None} * \cdots * ary[n-1] \mapsto \texttt{None}\end{aligned}$$

At this point, we have defined the *rep* function from Figure 7; it remains to define *rel*. A standard relation would be fine to implement this, but since each tree corresponds to exactly 1 finite map, we can simplify things by computing the finite map associated with the tree (using as_map) and stating that it equals the desired model. We can then pick the handle type to be a pointer and define the full representation predicate to be the conjunction of *rep*, *rel* and the pure facts about the tree structure.

$$\begin{aligned}&\texttt{repMap}\ hdl\ m \overset{\Delta}{\Longleftrightarrow} \exists h.\exists p : \texttt{ptree}\ h.hdl \mapsto (\texttt{ptrFor}\ p, \#p\#)\ *\\ &\quad \texttt{repTree}\ h\ \texttt{None}\ p\ *\ [m = \texttt{as\_map}\ p]\ *\ [\texttt{treeSorted}\ h\ p\ \texttt{Min}\ \texttt{Max}]\end{aligned}$$

By packing a copy of the ptree with the root pointer, we avoid the need to search for a model during proofs. The alternative is to show that there is at most one ptree that a given pointer and heap can satisfy (i.e., that repTree is *precise* [15]). However, this is complicated by the fact that the ptree type is indexed by the height. The pure treeSorted predicate combines all of the facts about the key constraints, but is not necessary for the iterator and was explained in previous work [12], so we do not explain it in detail.

With our representation for B+ trees, we can now turn to their iterators. Our approach is similar to the technique we applied to the list iterator. First, we state the heap predicate that divides the tree into the "trunk" and the branches as disjoint entities. We can achieve this with only minor discomfort by parameterizing repTree by the leaf case and passing the empty heap when we only want to describe the trunk. We also implement a function repLeaves to describe a list of leaves in isolation. These two functions satisfy the following property which is key to opening and closing our iterator:

$$\begin{aligned}&\forall h\ optr\ p.\ \texttt{repTree}\ optr\ p \Longleftrightarrow\\ &\quad \texttt{repTrunk}\ optr\ p * \texttt{repLeaves}\ (\texttt{Some}\ (\texttt{firstPtr}\ p))\ (\texttt{leaves}\ p)\ optr\end{aligned}$$

Using these predicates, we can define the representation of the iterator:

$$\begin{aligned}&\texttt{repIter}\ \mathit{own}\ h\ m\ \mathit{idx} \overset{\Delta}{\iff} \exists h.\exists \mathit{tr} : \texttt{ptree}\ h.\exists i.\exists \mathit{prev}.\exists \mathit{cur}.\exists \mathit{rest}.\\ &\quad \mathit{own} \mapsto (\texttt{ptrFor}\ \mathit{tr}, \#\mathit{tr}\#) * \texttt{repTrunk}\ h\ \texttt{None}\ p\ *\\ &\quad [m = \texttt{as\_map}\ p] * [\texttt{treeSorted}\ h\ p\ \texttt{Min}\ \texttt{Max}]\ *\\ &\quad h \mapsto (\mathit{cur}, i) * \texttt{repLeaves}\ \mathit{prev}\ \mathit{cur} * \texttt{repLeaves}\ \mathit{rest}\ \texttt{None}\ *\\ &\quad [\texttt{leaves}\ \mathit{tr} = \mathit{prev}\ \mathbin{+\!\!+}\ \mathit{rest}] * [\texttt{posInv}\ i\ \mathit{idx}\ \mathit{prev}\ \mathit{rest}\ m]\end{aligned}$$

The first two lines after the existentials correspond to the framed heap and pure facts needed to re-establish the tree representation invariant. The third line declares the iterator state ($h \mapsto (\mathit{cur}, i)$) and the combined `repLeaves` specify the representation of the leaves. Because each leaf could have a different number of key-value pairs, it is difficult to use the built-in `firstn` and `skipn` functions, so we existentially quantify two lists of leaves (*prev* and *rest*) and assert that their concatenation ($\mathbin{+\!\!+}$) must be equal to the leaves of the tree. The final pure fact establishes the invariant on *cur* and the index into the current leaf: if there are elements left to iterate, $i + \texttt{length}\,(\texttt{as\_map}\,\mathit{prev}) = \mathit{idx}$ and i is a valid index in the list. Otherwise, $i = 0$ and $\mathit{rest} = \texttt{nil}$.

6 Discussion

In this section we consider the overhead of verification (Section 6.1), summarize our sharing insights (Section 6.2), and review related work (Section 6.3).

6.1 The Burden of Mechanized Proofs

Our methodology places the burden of proof on the developer. Proof search scripts and lemmas are part of the final code and running them considerably increases compilation time. However, our proofs confirm functional (partial) correctness properties and our specifications document precise pre- and post-conditions for clients to use.

Figure 8, presents a quantitative look at the size of our development in number of lines. The *Spec* column counts command specifications; this is the interface that the client needs to reason about. Excluding the data structure invariants, this is the part of the code that a client of the library needs to reason about. The *Impl* column counts imperative code. The next two columns count auxiliary lemmas and automation. The first, *Sep. Lemmas*, counts lines that pertain to separation logic, while *Log. Lemmas* counts lines that only reason about pure structures, e.g. as lists. The *Overhead* column gives the ratio of proofs to specification and code. The *Time* column gives the time required to prove all of the verification conditions not including auxiliary lemmas. Line counts include only new lines needed for verifying the function, so, if a lemma is required for both `sub` and `insert` it is only counted against `sub`.

As Figure 8 shows, the first commands contribute the most to the proof burden because we are writing general lemmas about the model and representation

	Lines of Code				Overhead	
Command	Spec	Impl	Sep. Lemmas	Log. Lemmas	Lines	Time (m:s)
`new`	2	1	15	1	5.33x	0:00
`free`	3	13	33	0	2.06x	0:15
`insert`	9	25	15	11	0.76x	1:22
`delete`	9	26	1	7	0.23x	2:52
`sub`	3	14	1	0	0.06x	1:21
`mfold_left`	7	13	1	6	0.35x	1:47
`iterator`	3	3	29	0	4.83x	0:17
`close`	3	2	9	0	1.80x	0:11
`next`	3	8	30	13	3.91x	2:30
Total	82	123	155	73	1.11x	12:07

Fig. 8. Breakdown of lines of code for lists and iterators

predicate. Once these lemmas have been proven, the remainder of the commands are almost immediate. We believe that the logical lemmas required for our code are mostly within the capabilities of existing automated theorem provers [13] and integrating such tools would likely eliminate all of the overhead from this column. It is less likely that existing tools are directly applicable to our separation logic though existing automation is fairly good at this. The time spent interactively verifying our implementation was mostly spent abstracting lemmas which is straightforward but time consuming because of Coq's interaction model.

6.2 Sharing Lessons

While originally proposed for parallel code, fractional permissions for external sharing are important for sequential code. This is a by-product of multiple views of the same data structure, in our case lists and iterators. Our solution is simple because the list and iterator are completely decoupled and so we do not need to correlate mutation through multiple views[2]. Supporting mutation with a single iterator is relatively straightforward. The `ConcurrentModificationException` problem from Java is a general consequence of mutation of structures with multiple views and giving natural semantics to these operations is similar to the difficulty of writing precise specifications for concurrent functions.

When describing internal sharing, our technique allows us to specify equations directly on pointers. We find that quantifying all of the pointers at the beginning is useful for addressing this problem and it integrates nicely with our solution to heap structures being loosely related to interface models.This also allows us separately to state pure facts about the shape rather than having to fold them into the representation predicate. It is unclear how this technique could be applied to concurrent code, however, since this irrelevant, global state would need to be protected by a lock.

[2] The code for maintaining multiple concurrently mutable views is not simple, so the verification should not be expected to be trivial either.

6.3 Related Work

Weide [20] uses *model-oriented specification* in *Resolve* to specify how iterators behave. These specifications follow a *requires/ensures* template on top of a purely logical model, similar to Ynot's interface model.

Bierhoff [2] proposed using *type-state* specifications [10] for iterators. This system uses finite state machines to define the state of an object and specify when operations are permitted. This technique is particularly useful for specifying "non-interference" properties [19] such as marking a collection read-only when an iterator exists. We achieve this using fractional permissions, but can encode type states by adding a state parameter to the representation predicate of our data structures.

Our approach is most similar to the work of Krishnaswami [11] where separation and Hoare logic are combined to reason about iterators. His technique relies on the separating implication ($-\!\!*$), the separation logic analog of implication. We are interested in incorporating this into our separation logic, but we have not yet developed effective automation for it, so the burden of using it can be considerable. More recent work by Jensen [9] shows how a similar approach using separating implication can be applied to mutable views of a container.

B+ trees have been formalized in two previous developments. Bornat *et al.* [3] proposed using classical conjunction to capture the B+ tree as a tree and a list in the same heap. This is convenient for representation, but it requires re-establishing both the heap as a tree and as a list at every step of the code. By unifying the two views, we only need to reason about one view at a time. We support the two views by proving repTree is equivalent to a representation that exposes the leaves as a list.

Sexton and Thielecke [17] formulate B+ trees by defining a language of tree-operations for a stack-machine. Their representation is similar to our own in not using classical conjunction, but they quantify structure in the representation predicate which forces them to state the pure properties there as well.

7 Conclusions

In this work we have demonstrated a technique for building verified imperative software using theorem proving in the Ynot library for Coq.

We showed how external sharing can be achieved using abstract predicates which quantify over fractional permissions and showed how this technique can be applied to representing multiple views. Further, we showed how ownership types can be applied to make the view's representation predicate precise.

To address internal sharing we suggest simplifying recursive definitions by existentially quantifying all of the salient aspects of the data structure at the beginning of the representation predicate. This makes stating facts, such as aliasing equations, simple and allows the programmer to minimize the use of existential quantification which can be difficult to reason about automatically.

Future Work

The use of the separating implication in so many developments [11,17] demonstrates its usefulness. It would benefit our own development by allowing us to avoid duplicating parts of the representation predicate in the iterators. We are interested in extending Ynot's automation to reason about it and hope that doing so will reduce the burden of specifying and verifying Ynot code.

References

1. Aydemir, B.E., Bohannon, A., Fairbairn, M., et al.: Mechanized Metatheory for the Masses: The PoplMark Challenge. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 50–65. Springer, Heidelberg (2005)
2. Bierhoff, K.: Iterator specification with typestates. In: SAVCBS 2006, pp. 79–82. ACM, New York (2006)
3. Bornat, R., Calcagno, C., OHearn, P.: Local reasoning, separation and aliasing. In: Proceedings of SPACE, vol. 4 (2004)
4. Chlipala, A., et al.: Effective interactive proofs for higher-order imperative programs. In: ICFP 2009, pp. 79–90. ACM, New York (2009)
5. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: OOPSLA 1998, pp. 48–64. ACM, New York (1998)
6. Haack, C., Hurlin, C.: Separation Logic Contracts for a Java-Like Language with Fork/Join. In: Meseguer, J., Roşu, G. (eds.) AMAST 2008. LNCS, vol. 5140, pp. 199–215. Springer, Heidelberg (2008)
7. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969)
8. Boyland, J.: Checking Interference with Fractional Permissions, vol. 2694 (2003)
9. Jonas, B.J.: Specification and validation of data structures using separation logic. Master's thesis, Technical University of Denmark (2010)
10. Kim, T., Bierhoff, K., Aldrich, J., Kang, S.: Typestate protocol specification in JML. In: SAVCBS 2009. ACM, New York (2009)
11. Krishnaswami, N.R.: Reasoning about iterators with separation logic. In: SAVCBS 2006, pp. 83–86. ACM, New York (2006)
12. Malecha, G., Morrisett, G., Shinnar, A., Wisnesky, R.: Toward a verified relational database management system. In: POPL 2010 (January 2010)
13. De Moura, L., Bjrner, N.: Z3: An efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
14. Nanevski, A., Morrisett, G., Birkedal, L.: Polymorphism and separation in Hoare type theory. In: ICFP 2006. ACM, New York (2006)
15. O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theoretical Computer Science 375(1-3), 271–307 (2007)
16. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Symposium on Logic in Computer Science, LICS 2002 (2002)
17. Sexton, A., Thielecke, H.: Reasoning about B+ Trees with Operational Semantics and Separation Logic. Electronic Notes in Theoretical Computer Science 218, 355–369 (2008)
18. Sozeau, M., Oury, N.: First-class type classes. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 278–293. Springer, Heidelberg (2008)
19. Weide, B.W., et al.: Design and Specification of Iterators Using the Swapping Paradigm. IEEE Trans. Softw. Eng. 20(8), 631–643 (1994)
20. Weide, B.W.: SAVCBS 2006 challenge: specification of iterators. In: SAVCBS 2006, pp. 75–77. ACM, New York (2006)

Industrial-Strength Certified SAT Solving through Verified SAT Proof Checking

Ashish Darbari[1,*], Bernd Fischer[2], and João Marques-Silva[3]

[1] ARM, Cambridge, CB1 9NJ, England
[2] School of Electronics and Computer Science
University of Southampton, Southampton, SO17 1BJ, UK
[3] School of Computer Science and Informatics
University College Dublin, Belfield, Dublin 4, Ireland

Abstract. Boolean Satisfiability (SAT) solvers are now routinely used in the verification of large industrial problems. However, their application in safety-critical domains such as the railways, avionics, and automotive industries requires some form of assurance for the results, as the solvers can (and sometimes do) have bugs. Unfortunately, the complexity of modern and highly optimized SAT solvers renders impractical the development of direct formal proofs of their correctness. This paper presents an alternative approach where an untrusted, industrial-strength, SAT solver is plugged into a trusted, formally verified, SAT proof checker to provide industrial-strength certified SAT solving. The key characteristics of our approach are (i) that the checker is not tied to a specific SAT solver but certifies any solver respecting the agreed format for satisfiability and unsatisfiability claims, (ii) that the checker is automatically extracted from the formal development, and (iii) that the combined system can be used as a standalone executable program independent of any supporting theorem prover. The core of the system is a checker for unsatisfiability claims that is formally designed and verified in Coq. We present its formal design and outline the correctness criteria. The actual standalone checker is automatically extracted from the the Coq development. An evaluation of the checker on a representative set of industrial benchmarks from the SAT Race Competition shows that, albeit it is slower than uncertified SAT checkers, it is significantly faster than certified checkers implemented on top of an interactive theorem prover.

1 Introduction

Advances in Boolean satisfiability (SAT) technology have made it possible for SAT solvers to be routinely used in the verification of large industrial problems, including problems from safety-critical domains such as the railways, avionics, and automotive industries [11,16]. However, the use of SAT solvers in such domains requires some explicit form of assurance for the results since SAT solvers can and sometimes have bugs. For example, in the SAT 2007 competition, the solver SATzilla CRAFTED reported incorrect outcomes on several problems [20].

* This author was at the University of Southampton whilst this work was carried out.

Two alternative methods can be used to provide assurance. First, the solver could be proven correct once and for all, but this approach had limited success. For example, Lescuyer et al. [13] formally designed and verified a SAT solver using the Coq proof assistant [3], but without any of the techniques and optimizations used in modern solvers. Smith and Westfold [24] use a correct-by-construction approach to simultaneously derive code and correctness proofs for a family of SAT solvers, but their performance falls again short of the current state of the art. Reasoning about the optimizations makes the formal correctness proofs exceedingly hard. This was shown in the work of Marić [14], who verified at the pseudo-code level the algorithms used in the ARGO-SAT solver but did not verify the actual solver itself. In addition, the formal verification has to be repeated for every new SAT solver (or even a new version of a solver), or else users are locked into using the specific verified solver.

Alternatively, a *proof checker* can be used to validate each individual outcome of the solver independently; this requires the solver to produce a *proof trace* that is viewed as a certificate justifying its outcome. This approach was popularized by the Certified Unsatisfiable Track of the SAT 2007 competition [21] and was used in the design of several SAT solvers such as tts, booleforce, picosat, and zChaff. However, the corresponding proof checkers are typically implemented by the developers of the solvers whose output they check, which can lead to problems in practice. In fact, the checkers booleforce-res, picosat-res, and tts-rpt reported both "proof errors" and "program errors" on some of the benchmarks, although it is unclear what these errors signify.

Confidence can be increased if the checker is proven correct, once and for all. This is substantially simpler than proving the solver correct, because the checker is comparatively small and straightforward, and avoids system lock-in, because the checker can work for all solvers that can produce proof traces in the agreed format. This approach originates in the formal development of a proof checker for zChaff and Minisat proof traces by Weber and Amjad [26], and we follow it here as well. However, we depart considerably from Weber and Amjad in how we design and implement our solution. Their checker replays the derivation encoded in the proof trace *inside* an LCF-style theorem prover such as HOL 4 or Isabelle. Since the design and implementation of these provers relies on using the primitive inference rules of the underlying theorem prover, assurance is very high. However, their checker can run *only* inside the supporting prover, and not as a standalone tool, and performance bottlenecks become prominent when the size of the problems increases. Our checker, SHRUTI,[1] is formally designed and verified using the higher-order logic based proof assistant Coq [3], but we never use Coq for execution; instead we *automatically extract* an OCaml program from the formal development that can be compiled and used independently of Coq. This prevents the user from being locked-in to a specific proof assistant, and allows us to wrap SHRUTI around an industrial-strength but untrusted solver, to provide an industrial-strength certified solver that can be used as a regular component in a SAT-based verification work flow.

Our aim is not a fully formal end-to-end certification, which would in an extreme view need to include correctness proofs for file operations, the compiler, and even the

[1] SHRUTI in Sanskrit symbolizes 'knowledge' from a spoken word. In our case the outcome of our verified proof checker provides the knowledge about the correctness of a SAT solver and is therefore called SHRUTI.

hardware. Instead, we focus on the core of the checker, which is based on the resolution inference rule [18], and formally prove its design correct. We then rely on Coq's program extraction mechanism and some simple glue code as trusted components to build the entire checker. This way we are able to combine a high degree of assurance (much the same way as Amjad and Weber did) with high performance: as we will show in Section 4, SHRUTI is significantly (up to 32 times) faster than Amjad's checker implemented in HOL 4.

2 Propositional Satisfiability

2.1 Satisfiability Solving

Given a propositional formula, the goal of satisfiability solving is to determine whether there is an assignment of the Boolean truth values (i.e., true and false) to the variables in the formula such that the formula evaluates to true. If such an assignment exists, the given formula is said to be *satisfiable* or SAT, otherwise the formula is said to be *unsatisfiable* or UNSAT. Many problems of practical interest in system verification involve proving unsatisfiability, for example bounded model checking [5].

For efficiency purposes, SAT solvers represent the propositional formulas in conjunctive normal form (CNF), where the entire formula is a conjunction of *clauses*. Each clause itself denotes a disjunction of *literals*, which are simply (Boolean) variables or negated variables. An efficient CNF representation uses non-zero integers to represent literals. A positive literal is represented by a positive integer, whilst a negated one is denoted by a negative integer. Zeroes serve as clause delimiters. As an example, the (unsatisfiable) formula $(a \vee b) \wedge (\neg a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee \neg b)$ over two propositional variables a and b is thus represented in the widely used DIMACS notation as follows:

```
1 2 0      -1 2 0      1 -2 0      -1 -2 0
```

SAT solvers take a Boolean formula, and produce a SAT/UNSAT claim. A *proof-generating* SAT solver produces additional evidence (also called *certificates*) to support its claims. For a SAT claim, the certificate is simply an assignment, i.e., an enumeration of Boolean variables that need to be set to true in the input problem. It is trivial to check whether that assignment—and thus the original SAT claim—is correct: we simply substitute the Boolean values given by the assignment in the formula and then evaluate the overall formula, checking that it indeed is true. For UNSAT claims, the solvers return a resolution *proof trace* as certificate which is more complicated to check.

2.2 Proof Checking

When a solver claims a given problem is UNSAT and returns a proof trace as certificate, we can independently re-play the trace to check that its claim is correct: if we can follow the resolution inferences given in the trace to derive an empty clause, then we know that the problem is indeed UNSAT, and can conclude that the claim is correct.

A proof trace consists of the subset of the original input clauses used during resolution and the intermediate resolvents obtained by resolving the input clauses. The part of the proof trace that specifies how the input clauses have been resolved in sequence to derive a conflict (i.e., the empty clause) is organized as *chains*. These chains are also called regular input resolution proofs, or trivial proofs [2,4]. We call the input clauses in a chain its *antecedents* and its final resolvent simply its *resolvent*.

A key correctness constraint for the proof traces (and thus for the proof checker) is that whenever a pair of clauses is used for resolution, *at most* one complementary pair of literals is deleted, i.e., that the chains represent well-formed trivial resolution proofs [4]. Otherwise, we might erroneously "certify" an UNSAT claim for the satisfiable problem $(a \vee \neg b) \wedge (\neg a \vee b)$ by "resolving" over both complementary pairs of literals at once, to derive the empty clause. For efficiency reasons the chains are assumed to be ordered in such a way that we need to resolve only adjacent clauses, and, in particular, that *at least* one pair of complementary literals is deleted in each step. This allows us to avoid searching for the right clauses during checking, and to design a linear-time (in the size of the input clauses) algorithm.

2.3 PicoSAT Proof Representation

Most proof-generating SAT solvers [4,9,29] preserve the two criteria given above. We decided to work with PicoSAT [4] for three reasons. First, PicoSAT is efficient: it ranked as one of the best solvers in the industrial category of the SAT Competitions 2007 and 2009, and in the SAT Race 2008. Second, PicoSAT's proof representation is simple and records only the essential information. For example, it does not contain information about the pivot literals over which it resolves. Third, the representation is in *ASCII* format, which makes it easier to read and process than the more compact binary formats used by other solvers such as Minisat. Together the last two points help us simplify the design of SHRUTI and minimize the size of the trusted components outside the formal development.

A PicoSAT proof trace consists of rows representing the input clauses, followed by rows encoding the proof chains. Each row representing a chain consists of an asterisk (*) as place-holder for the chain's resolvent, followed by the identifiers of the clauses involved in the chain. Each chain row thus contains at least two clause identifiers, and denotes an application of one or more of the resolution inference rule, describing a trivial resolution derivation. Each row also starts with a non-zero positive integer denoting the identifier for that row's (input or resolvent) clause, and ends with a zero as delimiter. For the UNSAT formula shown in the previous section, the corresponding proof trace generated from PicoSAT looks as follows:

```
1  1  2 0     5 * 3 1 0
2 -1  2 0     6 * 4 2 5 0
3  1 -2 0
4 -1 -2 0
```

Rows 1 to 4 denote the input clauses from the original problem that are used in the resolution, with their identifiers referring to the original clause numbering, whereas rows 5 and 6 represent the proof chains. For example, in row 6 first the original clauses 4 and 2 are resolved and then the resulting clause is resolved against the resolvent from the previous chain, in total using two resolution steps.

By default, PicoSAT creates a compacted form of proof traces, where the antecedents for the derived clauses are not ordered properly within the chain. This means that there are instances in the chain where we "resolve" a pair of adjacent clauses but no literal is deleted. In this case we cannot deduce an existence of an empty clause for this trace unless we re-order the antecedents in the chain. However, PicoSAT comes with an uncertified proof checker called Tracecheck that can not only check the outcome of PicoSAT but also corrects this mis-ordering of traces. The outcome of Tracecheck is an extended proof trace and this then becomes the input to SHRUTI, i.e., we consider the combination of PicoSAT and Tracecheck as the solver to be checked. Hence, we can detect errors both in PicoSAT and Tracecheck's re-ordering algorithm, but do not distinguish them.

Similarly, it is possible to integrate other SAT solvers into SHRUTI, even if their proof traces use a different format, by developing a proof translator. This is usually straightforward [25]. As a proof of concept, we developed a translator from zChaff's proof format to PicoSAT's proof format. We again consider the combination of the core solver (i.e., zChaff), post-processor (i.e., the proof translator) and Tracecheck (used for extending the proof trace) as the system to be checked.

3 The SHRUTI System

3.1 High-Level Architecture

SHRUTI consists of a formally certified proof checker and a simple comparator that decides whether the solver's claim was correct. It takes as input a CNF file which contains the original problem description and a certificate (i.e., an assignment for a SAT claim or a resolution proof trace for an UNSAT claim). The checker evaluates the certificate and checks whether the two together denote a matching pair of SAT/UNSAT problem and solution. If this is the case, SHRUTI will accept the claim and output "yes", otherwise it will reject the claim and output "don't know". Note that the latter response does not imply that the solver's original claim is wrong—the problem may well be satisfiable or unsatisfiable as claimed. It only indicates that the given evidence (i.e., the assignment or the proof trace) is insufficient to corroborate the claim of the solver (i.e., the assignment does not evaluate to true, or the proof trace is not correct). This can happen due to mis-alignment of chains in the resolution proof as explained in Sect 2.3, or because the proof trace is not well-formed.

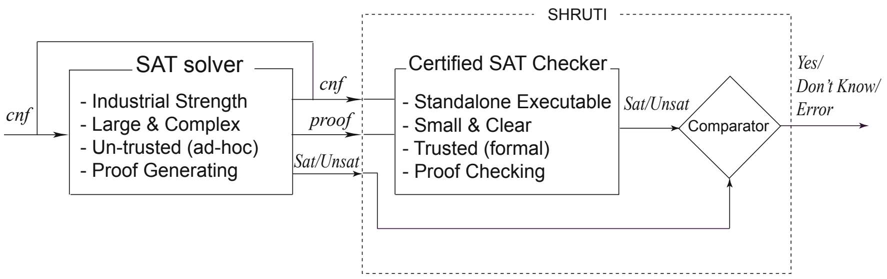


Fig. 1. SHRUTI's high-Level architecture

The crucial cases are where the problem is satisfiable (resp. unsatisfiable) but the solver claims the opposite, and produces a well-formed certificate for this wrong claim. SHRUTI contains a legitimacy check to prevent it from accepting forged certificates: it outputs "error" if it detects that the certificate is not legitimate, i.e., refers to variables (for a SAT claim) or clauses (for an UNSAT claim) that do not exist in the input problem. Hence, the only possibility under which SHRUTI would certify a wrong claim is if itself contained an error, but (accepting our characterization of the resolution function as correct) this is ruled out by our formal development. A high-level architectural view of our approach is shown in Figure 1.

We designed, formalized, and verified checkers for both satisfiability and unsatisfiability claims. In this paper, we focus on the more interesting aspect of checking unsatisfiability claims; satisfiability claims are significantly easier to check. The core component of the unsatisfiability checker is the development of the binary resolution inference rule inside the Coq proof assistant [3]. We show that the resolvent of a given pair of clauses is logically entailed by the two clauses (see Sect. 3.3), and that our implementation has the properties of the resolution inference rule [18,19] (see Sect. 3.4). In addition, we show that it maintains well-formedness of the clauses (see Sect. 3.5).

Once the formalization and proofs are complete, OCaml code is extracted from the development through the extraction API included in Coq. The extracted OCaml code expects its input in data structures such as tables and lists. These data structures are built by some glue code that also handles file I/O and pre-processes the proof traces (e.g., removes the zeroes used as separators for the clauses). Together with the comparator, the glue code is wrapped around the extracted checker and the result is then compiled to a native machine code executable that can be run independently of Coq.

```
let $checkUnsat(cnf, trace)$ = let $(clauses, chains) = split\ trace$ in
                               if $checkLegal(clauses, cnf)$
                               then  let $res = resolve(clauses, chains)$ in
                                     if $(res = [])$ then print "$yes$"
                                     else print "$don't\ know$"
                               else print "$error$"
```

The top-level function *checkUnsat* first splits the given trace into its constituent input clauses and the proof chains. It then checks whether the clauses used in the resolution proof are legitimate, i.e., whether all clauses used in the resolution proof trace are contained in the original problem CNF, or are derived by applying the resolution inference rule on legitimate clauses. Note that this checks only which clauses are used, not whether the result of an inference is correct. If the certificate is legitimate, then *resolve*, which is a wrapper around the formally verified binary resolution function, is used to derive the empty clause by re-playing the proof steps in the chains.

3.2 Formalization of Resolution in Coq

Coq is based on the Calculus of Inductive Constructions [7,8] and encapsulates the concepts of typed higher-order logic. It uses the notion of proofs as types, and allows constructive proofs and use of dependent types. It has been successfully used in the design and implementation of large scale certification of software such as in the CompCert [12] project. Our formal development in Coq follows the LCF style [22]; in particular, we only use definitional extensions, i.e., new theorems can only be derived by applying previously derived inference rules. We never use axiomatic extensions, which would allow us to assume the existence of a theorem without a proof, and thus invalidate the correctness guarantees of the extracted code.

In this section we present the formalization of SHRUTI in Coq. Its core logic is formalized as a shallow [1,17] embedding in Coq. In a shallow embedding we identify the object data types (i.e., the types used for SHRUTI) with the types of the meta-language (i.e., the Coq datatypes). Thus, inside Coq, we denote literals by integers, and clauses by lists of integers. Antecedents (denoting the input clauses) in a proof chain are represented by integers and a proof chain itself by a list of integers. We then define our resolution function to work directly on this integer-based representation.

The choice of a shallow embedding and the use of the integer-based representation were conscious design decisions, which make the internal data representation conceptually identical to the external problem representation. Consequently, our parsing functions can remain simple (and efficient), which minimizes the size of the trusted computing base. This is also a difference to Amjad's approach where external C++ functions were used for parsing and translating the integers into Booleans [27].

The main data structures that we used in the Coq formalization are lists and finite maps. The maps are used to represent resolution proofs internally. Their keys are the row identifiers obtained from the proof trace file. Their values are the actual clauses obtained by resolving the clauses specified in the proof trace—note that these are not part of the traces but must be reconstructed. When the trace is read, the identifier corresponding to the first proof chain becomes the starting point for checking. Once the resolvent is calculated for this, the process is repeated for all remaining chain rows, until we reach the end of the trace. If the entry for the last row is the empty clause, we conclude that the given problem and its trace represent an UNSAT instance.

We use the usual notation for quantifiers and logical connectives but distinguish implication over propositions ($\supset$) and over types ($\rightarrow$) for presentation clarity, though they are the same inside Coq. The notation $\Rightarrow$ is used during pattern matching (using match – with) as in other functional languages. For type annotation we use :, the set of integers is denoted by $\mathbb{Z}$, polymorphic lists by $list$ and list of integers by $list\, \mathbb{Z}$. The empty list is denoted by nil, and for the cons operation we use ::. List membership is represented by $\in$ and its negation by $\notin$. The function abs computes the absolute value of an integer. We use the keyword Definition to present our function definitions implemented in Coq but use let to define a function implemented directly in OCaml.

We define our resolution function ($\bowtie$) with the help of two auxiliary functions $union$ and $auxunion$. Both functions expect the input clauses to respect three well-formedness criteria: there should be no duplicates in the clauses ($NoDup$); there should be no complementary pair of literals *within* any clause ($NoCompPair$), and the clauses should be sorted by absolute value ($Sorted$). The first two assumptions are essentially the constraints imposed on input clauses when the resolution function is applied in practice. Sorting is enforced by us to keep our algorithm efficient. The predicate Wf encapsulates these properties.

Definition $Wf\ c = \ NoCompPair\, c\ \wedge\ NoDup\, c\ \wedge\ Sorted\, c$

Both *union* and *auxunion* use an accumulator to merge two clauses represented as sorted (by absolute value) integer lists, but differ in their behavior for complementary literals. *union* computes the resolvent by pointwise comparison of the literals. When it encounters a complementary pair of literals it *removes* both the complementary literals and calls *auxunion* to process the remainder of the lists. When *auxunion* encounters a complementary pair of literals it simply *copies* both the literals into the accumulator and recurses. Ideally, the proof traces contain only one pair of complementary literals for any pair of clauses that are resolved. However in reality, a solver or its proof trace can have bugs and it can create instances of clauses in the trace with multiple complementary pair of literals in a pair of clauses. Hence, we employ the two auxiliary functions to ensure that the resolution function deals with this in a sound way. Both functions also implement factoring, i.e., if they find the same literal in both clauses, only one copy is kept in the accumulator. Both functions also keep the accumulator sorted, which can be done by simply reversing, since all elements are in descending order.

```
Definition $c_1 \bowtie c_2$ = $union\ c_1\ c_2\ nil$

Definition $union\ (c_1\ c_2 : list\ \mathbb{Z})(acc : list\ \mathbb{Z})$ = match $c_1, c_2$ with
  | $nil, c_2 \Rightarrow app\ (rev\ acc)\ c_2$
  | $c_1, nil \Rightarrow app\ (rev\ acc)\ c_1$
  | $x :: xs, y :: ys \Rightarrow$ if $(x + y = 0)$ then $auxunion\ xs\ ys\ acc$
                              else if $(abs\ x < abs\ y)$ then $union\ xs\ (y :: ys)(x :: acc)$
                              else if $(abs\ y < abs\ x)$ then $union\ (x :: xs)\ ys\ (y :: acc)$
                              else $union\ xs\ ys\ (x :: acc)$
end

Definition $auxunion\ (c_1\ c_2 : list\ \mathbb{Z})(acc : list\ \mathbb{Z})$ = match $c_1, c_2$ with
  | $nil, c_2 \Rightarrow app\ (rev\ acc)\ c_2$
  | $c_1, nil \Rightarrow app\ (rev\ acc)\ c_1$
  | $x :: xs, y :: ys \Rightarrow$ if $(abs\ x < abs\ y)$ then $auxunion\ xs\ (y :: ys)\ (x :: acc)$
                              else if $(abs\ y < abs\ x)$ then $auxunion\ (x :: xs)\ ys\ (y :: acc)$
                              else if $x=y$ then $auxunion\ xs\ ys\ (x :: acc)$
                              else $auxunion\ xs\ ys\ (x :: y :: acc)$
end
```

3.3 Logical Characterization of the Resolution Function

The implementation of the checker is based on the operational characterization of the resolution inference rule, and in the next section, we will prove it correct with respect to this. However, we can also use the logical characterization of resolution, and prove the checker sound with respect to this. We need to prove that the resolvent of a given pair of clauses is logically entailed by the two clauses. Thus at an appropriate meta-level (since clauses are lists of non-zero integers, not Booleans), we need to prove a theorem of the following form $\forall c_1\, c_2\, c_3 \cdot\ (\{c_1, c_2\} \vdash_{\bowtie} c_3) \implies \{c_1, c_2\} \models c_3$.

Here, $\{c_1, c_2\} \vdash_{\bowtie} c_3$ denotes that c_3 is derivable from c_1 and c_2 using the resolution function $\bowtie$, and $\models$ denotes logical entailment. We can use the deduction theorem $\forall a\, b\, c \cdot \{a, b\} \models c \equiv (a \wedge b \implies c)$ and the fact that $\{c_1, c_2\} \vdash_{\bowtie} c_3$ is equivalent to

$c_1 \bowtie c_2 = c_3$ to re-state this as $\forall c_1\, c_2 \cdot (c_1 \wedge c_2) \implies (c_1 \bowtie c_2)$ which we prove its contrapositive form, $\forall c_1 c_2 \cdot \neg(c_1 \bowtie c_2) \implies \neg(c_1 \wedge c_2)$

In order to actually do this proof, we need to lift the interpretation of clauses and CNF from the level of the integer-based representation to the logical level. We thus define two evaluation functions $EvalClause$ and $EvalCNF$ that map an interpretation function I of type $\mathbb{Z} \rightarrow Bool$ over the underlying lists.

Definition $EvalClause\ nil\ I\ =\ False$
$EvalClause\ (x :: xs)\ I\ =\ I\ x\ \vee\ (EvalClause\ xs\ I)$

Definition $EvalCNF\ nil\ I\ =\ True$
$EvalCNF\ (x :: xs)\ I\ =\ (EvalClause\, x\ I)\ \wedge\ (EvalCNF\ xs\ I)$

Definition $Logical\ I = \forall (x : \mathbb{Z}) \cdot I(-x) = \neg(I\ x)$

The interpretation function must be logical in the sense that it maps the negation on the representation level to the negation on the logical level. With this, we can now state the soundness theorem that we proved.

Theorem 1. *(Soundness theorem)*
$\forall c_1 c_2 \cdot \forall I \cdot Logical\ I\ \supset\ \neg(EvalClause\ (c_1 \bowtie c_2)\ I)\ \supset\ \neg(EvalCNF\ [c_1, c_2]\ I)$

Proof. The proof proceeds by structural induction on c_1 and c_2. The first three sub-goals are easily proven by term rewriting and simplification by unfolding the definitions of $\bowtie$, $EvalClause$ and $EvalCNF$. The last sub-goal is proven by doing a case split on if-then-else and then using a combination of induction hypothesis and generating conflict among some of the assumptions. A detailed transcription of the Coq proof is available from http://www.darbari.org/ashish/research/shruti/. □

The soundness proof provides an explicit argument that the resolution function "does the right thing." This is different from Amjad and Weber's approach, who implemented their checker to work on the Bool representation of literals inside HOL and therefore relied on the implicit assurance obtained from using the inference rules of the HOL logic. They provide no explicit proof that their encoding is correct and soundness was never explicitly proven.

3.4 Correctness of the Resolution Function

In this section we prove that our implementation of the resolution function is operationally correct i.e., has the properties expected of the resolution function [18,19]. These properties can also be seen as steps towards a completeness proof, however, this is outside the scope of this paper. These are:

1. A pair of complementary literals is deleted in the resolvent obtained from resolving a given pair of clauses (Theorem 2).
2. All non-complementary pair of literals that are unequal are retained in the resolvent (Theorem 3).
3. For a given pair of clauses, if there are no duplicate literals within each clause, then for a literal that exists in both the clauses of the pair, only one copy of the literal is retained in the resolvent (Theorem 4).

For Theorem 2 to ensure that only a single pair of complementary literals is deleted we need to assume that there is a unique complementary pair ($\mathit{UniqueCompPair}$). The theorem will not hold in this form for the case with multiple complementary pairs.

Theorem 2. *A pair of complementary literals is deleted.*

$$\begin{aligned}&\forall c_1\ c_2 \cdot\ \mathit{Wf}\ c_1 \supset \mathit{Wf}\ c_2 \supset \mathit{UniqueCompPair}\ c_1\, c_2 \supset\\ &\quad \forall \ell_1\ \ell_2 \cdot\ (\ell_1 \in c_1) \supset (\ell_2 \in c_2) \supset (\ell_1 + \ell_2 = 0) \supset\\ &\qquad (\ell_1 \notin (c_1 \bowtie c_2)) \wedge (\ell_2 \notin (c_1 \bowtie c_2))\end{aligned}$$

In the following theorem, $\mathit{NoCompLit}\ \ell\, c$ asserts that the clause c contains no literal that is complementary to the given literal ℓ.

Theorem 3. *All non-complementary, unequal literals are retained.*

$$\begin{aligned}&\forall c_1\ c_2 \cdot\ \mathit{Wf}\ c_1 \supset \mathit{Wf}\ c_2 \supset\\ &\quad \forall \ell_1\ \ell_2 \cdot\ (\ell_1 \in c_1) \supset (\ell_2 \in c_2) \supset\\ &\qquad (\mathit{NoCompLit}\ \ell_1\, c_2) \supset (\mathit{NoCompLit}\ \ell_2\, c_1) \supset\\ &\qquad (\ell_1 \neq \ell_2) \supset (\ell_1 \in (c_1 \bowtie c_2)) \wedge (\ell_2 \in (c_1 \bowtie c_2))\end{aligned}$$

Our last correctness theorem is about factoring. We show that for equal literals in a given pair of clauses only one is copied in the resolvent.

Theorem 4. *Only one copy of equal literals is retained (factoring).*

$$\begin{aligned}&\forall c_1\ c_2 \cdot\ \mathit{Wf}\ c_1 \supset \mathit{Wf}\ c_2 \supset\\ &\quad \forall \ell_1\ \ell_2 \cdot\ (\ell_1 \in c_1) \supset (\ell_2 \in c_2) \supset (\ell_1 = \ell_2) \supset\\ &\qquad ((\ell_1 \in (c_1 \bowtie c_2)) \wedge (\mathit{count}\ \ell_1\, (c_1 \bowtie c_2) = 1))\end{aligned}$$

3.5 Preservation of Well-Formedness

Our implementation of the resolution function works correctly if the input clauses are well-formed. This implies that we prove that when we use the resolution function on a pair of well-formed clauses where there is only a single pair of literals to be resolved, we guarantee that the resolvent will be well-formed. This is shown in theorem below.

Theorem 5. *The resolvent of a pair of well-formed clauses is well-formed as well.*

$$\forall c_1\, c_2 \cdot\ \mathit{Wf}\ c_1 \supset \mathit{Wf}\ c_2 \supset \mathit{UniqueCompPair}\ c_1\ c_2 \supset \mathit{Wf}\ (c_1 \bowtie c_2)$$

Note that we assume the existence of a unique complementary pair of literals between the clauses c_1 and c_2 because the well-formedness only matters when the resolution function is applied on well-formed proof traces (i.e., one complementary pair of literals between any pair of clauses resolved).

3.6 Glue Code and Program Extraction

For the complete checker, we need to wrap a couple of auxiliary functions in Coq around the resolution function. These include *findAndResolve* which starts the checking process by first obtaining the clause identifiers from the proof trace file, and then invoking *findClause* to collect all the clauses for each row in the proof part of the proof trace file. A function called *checkResolution* recursively calls the function *findAndResolve* to apply the resolution function $\bowtie$ on each proof chain.

The top-level function in OCaml *checkUnsat* shown in Sect. 3.1 relies on the function *resolve*. This function (implemented in OCaml) first computes the number of proof steps from the chains (by counting the number of lines with an '*'), and then obtains the chains themselves and stores them in a table. This table is passed as an argument together with the number of proof steps and an empty table (to store resolvents) to the function *checkResolution* which calculates the resolvents for each step. Once the resolvent is obtained for the last row, its value is queried from the updated resolvent table and the value is returned as the final resolvent. These functions are implemented in OCaml directly because they handle file I/O, a feature not possible to implement inside Coq. An important observation is that the design of these OCaml functions though trivial is still necessary for using the core of the checker which is proven correct inside Coq.

We extract the OCaml code using the built-in extraction API in Coq. By default the extracted code would be implemented in terms of Coq datatypes. But this causes the implementation to be very inefficient at run time. A well-known technique [3] is to replace the Coq datatypes with equivalent OCaml datatypes. This is easily done by providing a mapping (between types) as an option when we do extraction. An important consequence of extraction is that only some datatypes, and data structures get mapped to OCaml's; the key logical functionality is unmodified. The decision for making changes in data types and data structures is a standard procedure used in any large-scale Coq related work such as the CompCert project [12]. For optimization purposes we thus made the following replacements:

1. Coq Booleans by OCaml Booleans.
2. Coq integers ($\mathbb{Z}$) by OCaml `int`.
3. Coq lists by OCaml lists.
4. Coq finite map by OCaml's finite map.
5. The combination of *app* and *rev* on lists in the function *union*, and *auxunion* was replaced by the tail-recursive List.rev_append in OCaml.

Replacing Coq's $\mathbb{Z}$ with OCaml integers gave a performance boost by a factor of 7-10. The largest integer (literal) we can denote depends on the choice of a 32-bit or a 64-bit OCaml `int`. The current mapping is done on a 32-bit signed integer; if SHRUTI encounters an integer greater than ± 2 billion (approx) it aborts with an error message. Making minor adjustments by replacing the Coq finite maps by OCaml ones and using tail recursive functions gave a further 20% improvement.

The Coq formalization consists of eight main function definitions amounting to 114 lines (not counting blank lines and comments), and the proofs of five main theorems shown in the paper and four more that are about maps (not shown here due to space limitations). The entire proof development is organized in several modules and is built interactively using the primitive inference rules of higher-order logic. The extracted code in OCaml is approximately 320 lines and the glue code implemented in OCaml is nearly 200 lines, including comments and print statements. The size of the extracted code is slightly larger than the original development in Coq because the Coq extractor produces code related to the libraries (integer, lists, and finite maps) used in our definitions. However, the actual size of the extracted code is not significant since it has been produced automatically using the extraction utility in Coq, which we believe to be correct much in the same way as we believe that the OCaml compiler and the underlying hardware are both correct.

4 Experimental Results

We evaluated SHRUTI on a set of industrial benchmarks from the SAT Races of 2006 and 2008 and the SAT Competition of 2007, and compared it to the Amjad and Weber's checkers that run inside the provers [28], and to the uncertified checker Tracecheck. We present our results on a sample of the SAT Race Benchmarks in Table 1. The results for SHRUTI shown in the table are for validating proof traces obtained from the PicoSAT solver. Our experiments were carried out on a server running Red Hat on a dual-core 3 GHz, Intel Xeon CPU with 28GB memory. Times shown for all the three checkers in the table are the total times including time spent on actual resolution checking, file I/O and garbage collection.

The HOL 4 and Isabelle checkers [28] were also evaluated on the SAT Race Benchmarks. The Isabelle-based version reported segmentation faults on most of the problems

Table 1. Comparison of our results with HOL 4 and Tracecheck

No.	Benchmark	HOL 4			SHRUTI			Tracecheck	
		Resolutions	*Time*	*inf/sec*	*Resolutions*	*Time*	*inf/s*	*Time*	*inf/s*
1.	een-tip-uns-numsv-t5.B	89136	4.61	19335	122816	0.86	142809	0.36	341155
2.	een-pico-prop01-75	205807	5.70	36106	246430	1.67	147562	0.48	513395
3.	een-pico-prop05-50	1804983	58.41	30901	2804173	20.76	135075	8.11	345767
4.	hoons-vbmc-lucky7	3460518	59.65	58013	4359478	35.18	123919	12.95	336639
5.	ibm-2002-26r-k45	1448	24.76	58	1105	0.004	276250	0.04	27625
6.	ibm-2004-26-k25	1020	11.78	86	1132	0.004	283000	0.04	28300
7.	ibm-2004-3_02_1-k95	69454	5.03	13807	114794	0.71	161681	0.35	327982
8.	ibm-2004-6_02_3-k100	111415	7.04	15825	126873	0.90	140970	0.40	317182
9.	ibm-2002-07r-k100	141501	2.82	50177	255159	1.62	157505	0.54	472516
10.	ibm-2004-1_11-k25	534002	13.88	38472	255544	1.77	144375	0.75	340725
11.	ibm-2004-2_14-k45	988995	31.16	31739	701430	5.42	129415	1.85	379151
12.	ibm-2004-2_02_1-k100	1589429	24.17	65760	1009393	7.42	136036	3.02	334236
13.	ibm-2004-3_11-k60	z?	z?	-	13982558	133.05	105092	59.27	235912
14.	manol-pipe-g6bi	82890	2.12	39099	245222	1.59	154227	0.50	490444
15.	manol-pipe-c9nidw_s	700084	26.79	26132	265931	1.81	146923	0.54	492464
16.	manol-pipe-c10id_s	36682	11.23	3266	395897	2.60	152268	0.82	482801
17.	manol-pipe-c10nidw_s	z?	z?	-	458042	3.06	149686	1.21	381701
18.	manol-pipe-g7nidw	325509	8.82	36905	788790	5.40	146072	1.98	398378
19.	manol-pipe-c9	198446	3.15	62998	863749	6.29	137320	2.50	345499
20.	manol-pipe-f6bi	104401	5.07	20591	1058871	7.89	134204	2.97	356522
21.	manol-pipe-c7b_i	806583	13.76	58617	4666001	38.03	122692	15.54	300257
22.	manol-pipe-c7b	824716	14.31	57632	4901713	42.31	115852	18.00	272317
23.	manol-pipe-g10id	775605	23.21	33416	6092862	50.82	119891	21.08	289035
24.	manol-pipe-g10b	2719959	52.90	51416	7827637	64.69	121002	26.85	291532
25.	manol-pipe-f7idw	956072	35.17	27184	7665865	68.14	112501	30.74	249377
26.	manol-pipe-g10bidw	4107275	125.82	32644	14776611	134.92	109521	68.13	216888

[27], but results for the HOL 4 implementation are summarized along with ours in Table 1. The symbol z? denotes that the underlying zChaff solver timed out after an hour. Since we were unable to get the HOL 4 implementation working on our system, it was run on a (comparable) AMD dual-core 3.2 GHz processor running Ubuntu with 4GB of memory. Amjad reported that the version of the checker he has used on these benchmarks is much faster than the one published in [28]. Since Amjad's work is based on proof traces obtained from ZVerify, the uncertified checker for zChaff, the actual proof traces checked by the HOL 4 implementation differ substantially from those checked by SHRUTI. We thus compare the speed in terms of resolution steps (i.e., inferences) checked per second, and observe that SHRUTI is 1.5 to 32 times faster than HOL 4. In addition, as a proof of concept we also validated the proof traces from zChaff by translating them to PicoSAT's trace format. The performance of SHRUTI in terms of inferences per second on the translated proof traces (from zChaff to PicoSAT) was similar to the performance of SHRUTI when it checked PicoSAT's traces obtained directly from the PicoSAT solver—something that is to be expected. We also compare our timings with that obtained from the uncertified checker Tracecheck; here, SHRUTI is about 2.5 times slower, on exactly the same proof traces.

We noticed that OCaml's native code compilation produces efficient binaries but the default settings for automatic garbage collection were not useful, and for large proof traces it ended up consuming (and thereby delaying the overall computation) as much as 60% of the total time. By increasing the initial size of major heap and making the garbage collection less eager, we reduced the computation times of our checker by almost an order of magnitude on proof traces with over one million inferences.

5 Related Work

Recent work on checking the result of SAT solvers can be traced to the work of Zhang and Malik [29] and Goldberg and Novikov [10], with additional insights provided in recent work [2,25]. The work closest to ours is that by Amjad and Weber, which we have already discussed throughout the paper. Bulwahn et al. [6] also have advocated the use of a checker, and experimented with the idea of reflective theorem proving in Isabelle, suggesting that it can be used for designing a SAT checker. However, no performance results were given. Shankar [23] proposed an approach generally based on a verified SAT solver, for checking a variety of checkers.

Marić [14], presented a formalization in Isabelle of SAT solving algorithms that are used in modern day SAT solvers. An important difference is that while we have formalized a SAT checker and *extracted* an executable code from the formalization itself, Marić formalizes a SAT solver (at the abstract level of state machines) and then implements the verified algorithm in the SAT solver *off-line*.

An alternative line of work involves the formal development of SAT solvers. Lescuyer and Conchon [13] have formalized a simplified SAT solver in Coq and extracted an executable. However, they have not formalized several of the key techniques used in modern SAT solvers, and have not reported performance results on any industrial benchmarks. The work of of Smith and Westfold [24] involves the formal synthesis of a SAT solver from a high level description. Albeit ambitious, this work does not include the most effective techniques used in modern SAT solvers.

There has also been interest in the area of certifying SMT solvers. M. Moskal recently provided an efficient certification technique for SMT solvers [15] using term-rewriting systems. The soundness of the proof checker is guaranteed through a formalization using inference rules provided in a term-rewriting formalism.

6 Conclusion

In this paper we presented a methodology for performing efficient yet formally certified SAT solving. The key feature of our approach is that we can combine a formally designed and verified proof checker with industrial-strength SAT solvers such as PicoSAT and zChaff to achieve industrial-strength certified SAT solving. We used the Coq proof-assistant for the formal development, but relied on its program extraction mechanism to obtain an OCaml program which was used as a standalone executable to check the outcome of the solvers. Any proof generating SAT solver that supports the PicoSAT's proof format can be plugged directly into our checker; different formats require only a simple proof translation step.

On the one hand, our checker provides much higher assurance compared to uncertified checkers such as Tracecheck and on the other it enhances usability and performance over certified checkers implemented inside provers such as HOL 4 and Isabelle. In this regard our approach provides an arguably optimal middle ground between the two extremes. We believe that such verified result checkers can be developed for other problem classes as well, and that this is a viable approach to verified software development. We are investigating on optimizing the overall performance of our checker even further so that the slight difference to uncertified checkers can be further minimized. We are also investigating checking SMT proofs.

Acknowledgements. We thank H. Herbelin, Y. Bertot, P. Letouzey, and other people on the Coq mailing list who helped us with Coq questions. J. Harrison gave useful suggestions on the soundness proof. We also thank T. Weber and H. Amjad for answering our questions on their work and also carrying out industrial benchmark evaluation on their checker. This work was partially funded by EPSRC Grant EP/E012973/1, and EU Grants ICT/217069 and IST/033709.

References

1. Angelo, C.M., Claesen, L., De Man, H.: Degrees of formality in shallow embedding hardware description languages in HOL. In: Joyce, J.J., Seger, C.-J.H. (eds.) HUG 1993. LNCS, vol. 780, pp. 89–100. Springer, Heidelberg (1994)
2. Beame, P., Kautz, H.A., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. J. Artif. Intell. Res. 22, 319–351 (2004)
3. Bertot, Y., Castéran, P.: Interactive theorem proving and program development. Coq'Art: The calculus of inductive constructions (2004)
4. Biere, A.: PicoSAT essentials. J. Satisfiability, Boolean Modeling and Computation 4, 75–97 (2008)
5. Biere, A., Cimatti, A., Clarke, E., Strichman, O., Zhu, Y.: Bounded Model Checking. In: Advances in Computers. Academic Press, London (2003)
6. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 134–149. Springer, Heidelberg (2008)

7. Coquand, T., Huet, G.: The Calculus of Constructions. Inf. Comput. 76(2-3), 95–120 (1988)
8. Coquand, T., Paulin, C.: Inductively defined types. In: Martin-Löf, P., Mints, G. (eds.) COLOG 1988. LNCS, vol. 417, pp. 50–66. Springer, Heidelberg (1990)
9. Een, N., Sorensson, N.: An extensible sat-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
10. Goldberg, E.I., Novikov, Y.: Verification of proofs of unsatisfiability for CNF formulas. In: Proc. Design, Automation and Test in Europe, pp. 10886–10891. IEEE, Los Alamitos (2003)
11. Hammarberg, J., Nadjm-Tehrani, S.: Formal verification of fault tolerance in safety-critical reconfigurable modules. J. Software Tools for Technology Transfer 7(3), 268–279 (2005)
12. Leroy, X., Blazy, S.: Formal Verification of a C-like Memory Model and its uses for Verifying Program Transformations. J. Automated Reasoning 41(1), 1–31 (2008)
13. Lescuyer, S., Conchon, S.: A reflexive formalization of a SAT solver in Coq. In: Proc. Emerging Trends of TPHOLs (2008)
14. Marić, F.: Formalization and implementation of modern SAT solvers. J. Automated Reasoning 43(1), 81–119 (2009)
15. Moskal, M.: Rocket-fast proof checking for SMT solvers. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 486–500. Springer, Heidelberg (2008)
16. Penicka, M.: Formal approach to railway applications. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems. LNCS, vol. 4700, pp. 504–520. Springer, Heidelberg (2007)
17. Boulton, R., Gordon, A., Gordon, M.J.C., Herbert, J., van Tassel, J.: Experience with embedding hardware description languages in HOL. In: Proc. of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience, pp. 129–156. North-Holland, Amsterdam (1992)
18. Robinson, J.A.: A Machine-Oriented Logic Based on the Resolution Principle. J. ACM 12(1), 23–41 (1965)
19. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach, 2nd edn. Prentice Hall, Englewood Cliffs (2003)
20. SAT 2007 Competition (2007), http://www.cril.univ-artois.fr/SAT07/results/globalbysolver.php?idev=11&det=1
21. SAT 2007 Competition - Phase 2 (2007), http://users.soe.ucsc.edu/ avg/ProofChecker/cert-poster-sat07.pdf
22. Scott, D.S.: A type-theoretical alternative to ISWIM, CUCH, OWHY. Theor. Comput. Sci. 121(1-2), 411–440 (1993)
23. Shankar, N.: Trust and Automation in Verification Tools. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 4–17. Springer, Heidelberg (2008)
24. Smith, D.R., Westfold, S.J.: Synthesis of propositional satisfiability solvers. Technical report, Kestrel Institute (April 2008)
25. Van Gelder, A.: Verifying propositional unsatisfiability: Pitfalls to avoid. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 328–333. Springer, Heidelberg (2007)
26. Weber, T.: Efficiently checking propositional resolution proofs in Isabelle/HOL. In: 6th Intl. Workshop Implementation of Logics, Phnom Penh 2006 (2006)
27. Weber, T., Amjad, H.: Private communication
28. Weber, T., Amjad, H.: Efficiently checking propositional refutations in HOL theorem provers. J. Applied Logic 7(1), 26–40 (2009)
29. Zhang, L., Malik, S.: Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In: Proc. Design, Automation and Test in Europe, pp. 10880–10885. IEEE, Los Alamitos (2003)

Dynamite 2.0: New Features Based on UnSAT-Core Extraction to Improve Verification of Software Requirements

Mariano M. Moscato[1], Carlos G. López Pombo[1], and Marcelo F. Frias[2]

[1] Department of Computer Science, FCEyN,
Universidad de Buenos Aires and CONICET
{mmoscato,clpombo}@dc.uba.ar

[2] Department of Computer Engineering,
Technological Institute of Buenos Aires (ITBA) and CONICET
mfrias@itba.edu.ar

Abstract. According to the Verified Software Initiative manifesto,

"*Lightweight techniques and tools have been remarkably successful in finding bugs and problems in software. However, their success must not stop the pursuit of this projects long-term scientific ideals*".

The Dynamite Proving System (DPS) blends the good qualities of the lightweight formal method Alloy with the certainty provided by the theorem prover PVS. Using the Alloy Analyzer during the proving process improves the PVS theorem proving experience by reducing the number of errors introduced along creative proof steps. Therefore, rather than becoming an obstacle to the goals of the Initiative, inside DPS Alloy becomes an aid. In this article we introduce new features of DPS based on the novel use of unsat cores to guide the proving process by pruning unnecessary information. We illustrate these new features using a non-trivial case-study coming from the networking domain.

1 Introduction

The Dynamite Proving System (DPS) was presented in [7]. The rationale behind DPS is that automated analysis, albeit incomplete, should support formal verification processes based on theorem proving. DPS has Alloy [9] as its specification language. Alloy's syntax includes constructs ubiquitous in modern object-oriented languages. The Alloy Analyzer [10], an analysis tool that provides automated (partial) analysis of Alloy specifications, makes Alloy a lightweight formal method with increasing adoption in academy and industry. The Alloy language (an extension of first-order logic with reflexive-transitive closure) is quite appropriate for modeling critical systems. According to the VSI manifesto [8],

"*Even though software requirements cannot be verified against a customers informal needs and desires, a great deal of clarity, insight, and precision can be gained by formalizing these requirements as a more precise specification. Once this is done, verification technology can be applied* *to the resulting formal specification, to investigate its consistency and to see if it captures important system properties such as safety or security.*"

The partial analysis offered by the Alloy Analyzer assumes data domains have a user-bounded size, called *scope*. Moreover, the analysis technique employed by the Alloy Analyzer (based on SAT-solving), does not scale well enough when domain sizes are increased past a (specification dependent) threshold. This should not be considered a shortcoming. Alloy's use is targeted at model debugging, and therefore small domain sizes are many times sufficient to uncover errors in specifications. Unfortunately, this kind of analysis only allows us to conclude that a system model is consistent, safe or secure up to a given size for data domains.

The Dynamite Proving System was developed with the intention of providing a tool for the verification (in the sense of the VSI Manifesto) of Alloy models. In order to accomplish this task, DPS extends the PVS [12] semi-automated theorem prover with a complete calculus for Alloy. We also integrated the Alloy Analyzer into DPS in order to automatically detect bugs introduced during creative proof steps (introduction of lemmas, new hypotheses, etc.) Including a pretty printer that exhibits sequents using Alloy notation, DPS provides the clerical Alloy user a more amenable and less error-prone theorem proving experience.

In order to better convey the contributions of this article, we will briefly discuss the proving process within PVS and DPS. DPS provides a complete calculus for Alloy, implemented on top of the higher-order calculus provided by PVS. In order to prove that a set of formulas $\Delta = \{\delta_1, \ldots, \delta_m\}$ follows from a set of hypotheses $\Gamma = \{\gamma_1, \ldots, \gamma_k\}$, one begins with the *sequent* $\Gamma \vdash \Delta$. Applying *inference rules*, from $\Gamma \vdash \Delta$ one must reach other sequents that can be recognized as *valid* (for example, sequents of the form $\alpha \vdash \alpha$). The informal understanding of the sequent $\Gamma \vdash \Delta$ is that from the conjunction of the formulas in Γ, the disjunction of the formulas in Δ must follow. The formulas in Γ (Δ) are called the *antecedent* (*consequent*) of the sequent.

The application of an inference rule results in one or more new sequents whose proofs provide a proof of the original sequent. Therefore, proofs in this kind of calculi are usually seen as *trees* in which the root is the sequent $\Gamma \vdash \Delta$. When all the leaves of the proof tree are *valid* sequents (in the sense mentioned before) the tree is considered *closed* and the proof is finished. In Fig. 1 we present, as examples, proof rules in order to deal with conjunctions in the antecedent and the consequent, respectively.

On start of a proof of an Alloy assertion α, DPS presents sequent $\vdash \alpha$. A proof must then be derived using the inference rules. Whenever the application

$$\frac{\alpha, \beta, \Gamma \vdash \Delta}{\alpha \wedge \beta, \Gamma \vdash \Delta} \wedge\vdash \qquad\qquad \frac{\Gamma \vdash \Delta, \alpha \qquad \Gamma \vdash \Delta, \beta}{\Gamma \vdash \alpha \wedge \beta, \Delta} \vdash\wedge$$

Fig. 1. Proof rules for conjunction

of an inference rule introduces new goals (sequents) to be proved, some of the antecedents and consequents inherited by the new sequents may be unnecessary to close the branch that initiates in that sequent. Our experience using DPS in our case-study is that along a proof of a given assertion the number of antecedents and consequents in intermediate sequents tends to grow. This leads many times to formulas that are not necessary in order to prove the sequents. These formulas make the identification of new proof steps more complex. PVS provides a command (`hide`) for hiding hypotheses and conclusions in sequents, yet its use is error-prone: removing necessary antecedents or consequents makes the proof infeasible.

In this article we will use an Alloy UnSAT-core [16] in order to remove formulas from sequents and from the underlying theories. An Alloy UnSAT-core is a subset of formulas (and even parts of formulas) from an inconsistent (up-to the selected scopes) Alloy theory that is itself inconsistent. How is an inconsistent theory obtained at a given point in the proving process? Notice that proving a sequent $\Gamma \vdash \Delta$ in a theory Ω (where $\Gamma = \{\gamma_1, \ldots, \gamma_k\}$ and $\Delta = \{\delta_1, \ldots, \delta_m\}$), is equivalent to proving in theory Ω the sequent

$$\vdash \left(\bigwedge_{1 \le i \le k} \gamma_i \right) \Rightarrow \left(\bigvee_{1 \le j \le n} \delta_j \right) .$$

Elemental logic reasoning allows us to conclude that the former sequent is derivable if and only if the theory

$$\Omega \cup \left\{ \neg \left(\left(\bigwedge_{1 \le i \le k} \gamma_i \right) \Rightarrow \left(\bigvee_{1 \le j \le n} \delta_j \right) \right) \right\}$$

is inconsistent.

Notice that since we are not requesting a *minimal* UnSAT-core, the UnSAT-core might be the whole theory. Fortunately, this is most times not the case.

The contributions of this article are summarized as follows:

1. We release DPS 2.0, downloadable from `http://www.dc.uba.ar/dynamite`.
2. We present a novel heuristic to reduce the proof search space based on the use of UnSAT cores to remove possibly unnecessary antecedents and consequents in a sequent. The technique also allows us to remove formulas from the underlying theories.
3. We discuss the presented heuristic focusing on its (un)soundness and/or (in)completeness.
4. We discuss the applicability of the heuristic using as a reference a nontrivial case-study.
5. We present some experimental results obtained from extensively using this heuristic along the proving process in our case-study.

The article is organized as follows. In Section 2 we present our running example. In Section 3 we describe the proving process within Dynamite, including a discussion on how lightweight and heavyweight formal methods can be combined in a synergic way. In Section 4, after a short introduction to UnSat-cores, we present our heuristic for proof space reduction and discuss some experiences learnt using the technique. In Section 5 we discuss related work. Finally, in Section 6 we present our conclusions and proposals for further work.

2 Compositional Binding in Network Domains

In order to test the usefulness of the techniques we will present in this article, we worked on an Alloy model presented by Zave in [21]. There are good reasons for choosing this model. The first one is that although the model is not extremely complex, its analysis using the Alloy Analyzer does not scale for some properties even for small scopes. Notice that "small scope" is a subjective notion that strongly depends on the user knowledge about the model. It is seldom the case that Alloy models include information on the willingness of the user to analyze the model for scopes that surpass the possibilities of the Alloy Analyzer. This model is particular in that it thoroughly documents the intentions of the user:

```
check StructureSufficientForPairReturnability for 2 but
   2 Domain, 2 Path, 3 Agent, 9 Identifier -- this one is too big also
check StructureSufficientForPairReturnability for 2 but
   2 Domain, 2 Path, 3 Agent, 11 Identifier
-- attempted but not completed at MIT; formula is not that large; results
-- suggest that the problem is very hard, and that the formula is almost
-- certain unsatisfiable [which means that the assertion holds]
```

Notice that the modeler was concerned enough about the validity of the model assertions that she requested assistance from the developers of the Alloy Analyzer. The limitations of the automated techniques open the possibility to use verification in order to determine the validity (or not) of the model assertions. Of course there are other Alloy models that are also good candidates to be verified using DPS. Among these, we want to mention the Mondex electronic purse presented in [14], or the Flash filesystem presented in [11]. Since these problems have become sort of benchmarks for different analysis and verification techniques, it was our intention to leave them as interesting case-studies that might attract new users of DPS.

Zave formalizes a mechanism for binding of identifiers in the delivery of messages in computer networks. Thus, the model is not a specification of an isolated software or hardware artifact but rather the specification of network services whose implementation may involve several software and hardware artifacts. This model is mainly about communication in computer networks, and, more specifically, about how communicating agent identifiers are bound so that the messages reach their correct destination. Properties about the possibility of reaching an agent, determinism in the delivery of messages, existence of cycles in the routing of messages and the possibility of constructing a return path for a message are formally specified in the model. In particular, the model studies how these properties are affected by the addition of new bindings of identifiers.

Communicating artifacts in these kinds of networks may be software systems or hardware devices. As this distinction is not important for the specification, all the communicating artifacts are called *agents*. Thus, the communications are established between agents and take place over network domains.

An agent g is considered *reachable* in a domain d from an identifier i if:

- i is connected to an address a in the reflexive and transitive closure of the binary relation formed by all the bindings corresponding to d,
- a cannot be bound to another identifier in d, and
- a can route messages to g in d.

Figure 2 shows an Alloy assertion `BindingPreservesReachability`. This assertion states that if an agent is reachable in a domain d, it is also reachable in the domain resulting from adding a new binding to d, provided that the newly bound identifiers are not used in d. This latter condition is formalized by a predicate `IdentifiersUnused`.

```
assert BindingPreservesReachability {
   all d, d': Domain, newBinding: Identifier -> Identifier |
   IdentifiersUnused(d,newBinding.Identifier) &&
   AddBinding(d,d',newBinding)
   => (all i: Identifier, g: Agent |
         ReachableInDomain(d,i,g) => ReachableInDomain(d',i,g) ) }
```

Fig. 2. One proved property: `BindingPreservesReachability`

A domain is called *deterministic* if each identifier is associated to at most one agent. One of the properties to be analyzed for this model states that

> *whenever a new binding for an unused identifier is added to a deterministic domain, it remains deterministic.*

A domain is considered *non-looping* if the transitive closure of the bindings for that domain has no cycles. A second assertion then states that

> *the addition of a new binding to a non-looping domain does not change this condition whenever the transitive closure of the new binding does not have cycles.*

Also a notion of *structured* domain is introduced.

In [21], Zave used the Alloy Analyzer to analyze this model and concluded that these five properties hold for Alloy domains containing at most two network domains and four elements in each set (such as identifiers, agents, etc).

Using DPS we have proved that the following assertions hold despite their domain bounds:

– `BindingPreservesReachability`,
– `BindingPreservesDeterminism`,
– `BindingPreservesNonLooping`,
– `ABindingPreservesStructure`, and
– `BBindingPreserverStructure`.

Notice that these assertions suffer the similar limitations, regarding their analyzability, with assertion `StructureSufficientForPairReturnability`.

3 An Introduction to the Dynamite Proving System

The Dynamite Proving System is an extension of the PVS theorem prover [12] that interacts with the Alloy Analyzer. Alloy is a formal modeling language well suited for modeling of critical systems. Its simple semantics based on relations and the automated analysis provided by the Alloy Analyzer make Alloy an increasingly accepted lightweight formal method. The analysis provided by the Alloy Analyzer assumes domains sizes are user-bounded, and is therefore partial. This makes the Alloy Analyzer unsuitable for verification of critical models. An alternative would be the use of a theorem prover in order to verify Alloy assertions. Unfortunately, no complete calculus for Alloy was known. In [7] we presented such complete calculus, and extended PVS in order to include the calculus. An appropriate pretty-printer allowed us to present formulas using Alloy notation.

While PVS automatically detects syntactic errors and uses proof techniques in order to (try to) automatize parts of the proofs, some errors many times cannot be detected. We refer to the errors that occur when:

1. An invalid sequent has to be proved.
2. An invalid lemma is introduced.
3. A new hypothesis (which does not follow from the axioms in the current model or the antecedents of the sequent being proved) is added.
4. A necessary formula is incorrectly hidden from a sequent.

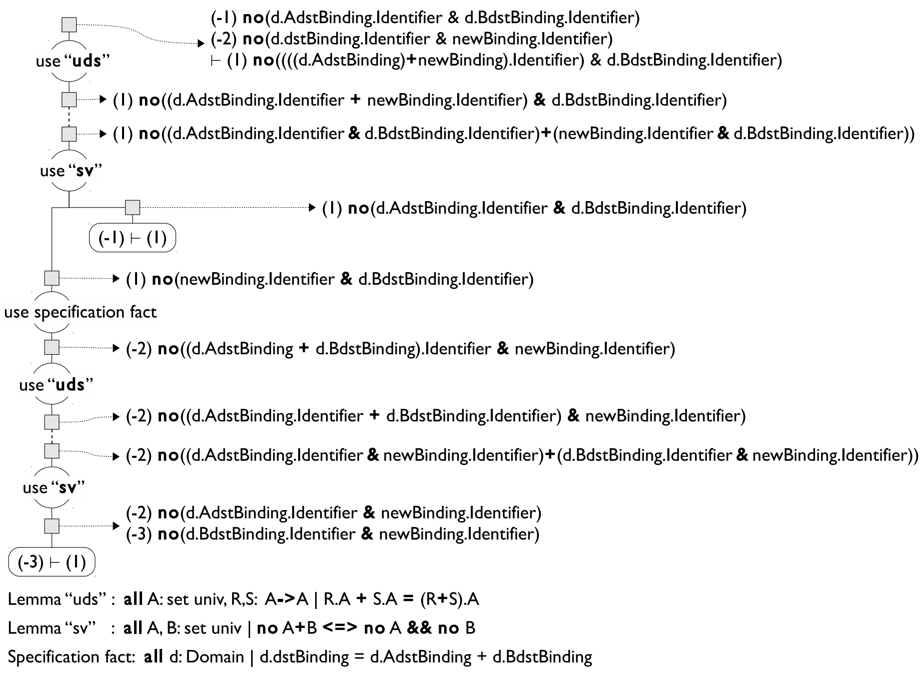
In [7] we deal with the first three situations. In order to reduce the chances of introducing erroneous lemmas or hypotheses, DPS resorts to the Alloy Analyzer. Let us assume we are proving a sequent $\gamma_1, \dots, \gamma_k \vdash \delta_1, \dots, \delta_n$, and a new hypothesis φ is introduced using the PVS command `(case varphi)`. According to PVS, we are now left with two sequents to prove, namely,

$$\gamma_1, \dots, \gamma_k, \varphi \vdash \delta_1, \dots, \delta_n \quad \text{and} \quad \gamma_1, \dots, \gamma_k \vdash \delta_1, \dots, \delta_n, \varphi.$$

It might be the case that φ is overly strong, i.e., it simplifies proving sequent $\gamma_1, \dots, \gamma_k, \varphi \vdash \delta_1, \dots, \delta_n$, but sequent $\gamma_1, \dots, \gamma_k \vdash \varphi$ (which allows us to discharge the newly added hypothesis) is not valid. In order to detect such situations, an Alloy model is automatically created. The model contains, as an assertion to be checked using the Alloy Analyzer, the formula

$$\left(\bigwedge_{1 \le i \le k} \gamma_i \right) \Rightarrow \varphi.$$

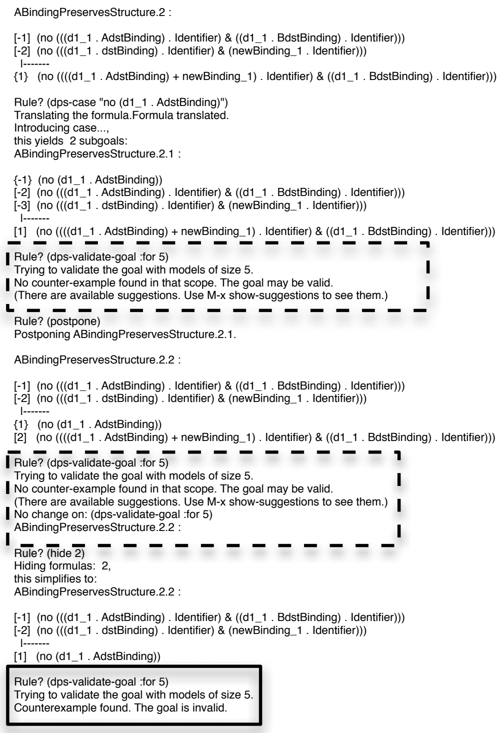
If a counterexample is returned by the Alloy Analyzer, then it is automatically reported by DPS. The existence of a counterexample means that the sequent is not valid and therefore formula φ is too restrictive. In Fig. 3 we show a proof fragment from our case-study where this happens. We have a sequent S of the form $\gamma_1, \gamma_2 \vdash \delta_1$. We then introduce a new hypothesis h, and obtain new goals $\gamma_1, \gamma_2, h \vdash \delta_1$ and $\gamma_1, \gamma_2 \vdash \delta_1, h$. The proof structure for S is:


(-1) **no**(d.AdstBinding.Identifier & d.BdstBinding.Identifier)
(-2) **no**(d.dstBinding.Identifier & newBinding.Identifier)
⊢ (1) **no**((((d.AdstBinding)+newBinding).Identifier) & d.BdstBinding.Identifier)
use "**uds**"
(1) **no**((d.AdstBinding.Identifier + newBinding.Identifier) & d.BdstBinding.Identifier)
(1) **no**((d.AdstBinding.Identifier & d.BdstBinding.Identifier)+(newBinding.Identifier & d.BdstBinding.Identifier))
use "**sv**"
(1) **no**(d.AdstBinding.Identifier & d.BdstBinding.Identifier)
(-1) ⊢ (1)
(1) **no**(newBinding.Identifier & d.BdstBinding.Identifier)
use specification fact
(-2) **no**((d.AdstBinding + d.BdstBinding).Identifier & newBinding.Identifier)
use "**uds**"
(-2) **no**((d.AdstBinding.Identifier + d.BdstBinding.Identifier) & newBinding.Identifier)
(-2) **no**((d.AdstBinding.Identifier & newBinding.Identifier)+(d.BdstBinding.Identifier & newBinding.Identifier))
use "**sv**"
(-2) **no**(d.AdstBinding.Identifier & newBinding.Identifier)
(-3) **no**(d.BdstBinding.Identifier & newBinding.Identifier)
(-3) ⊢ (1)
Lemma "uds" : **all** A: set univ, R,S: A->A | R.A + S.A = (R+S).A
Lemma "sv" : **all** A, B: set univ | **no** A+B **<=>** **no** A **&&** **no** B
Specification fact: **all** d: Domain | d.dstBinding = d.AdstBinding + d.BdstBinding

Notice the following:

– Goals `ABindingPreservesStructure.2.1` and `2.2` are validated in Fig. 3 using the Alloy Analyzer. Notice that no counterexamples are found (as reported inside the dashed boxes), and therefore the goals may be correct.
– When goal `ABindingPreservesStructure.2.2` is validated *after* formula 2 is hidden, inside the solid square a counterexample is reported. Notice that hiding formula 2 is a reasonable decision, since we are trying to verify that the introduced hypothesis indeed follows from the sequent antecedents.
– Although not related to the technique, we want to stress the fact that formulas in sequents are actual Alloy formulas.

The counterexample can be used in order to gain a better understanding of the model.



```
ABindingPreservesStructure.2 :

[-1]  (no (((d1_1 . AdstBinding) . Identifier) & ((d1_1 . BdstBinding) . Identifier)))
[-2]  (no (((d1_1 . dstBinding) . Identifier) & (newBinding_1 . Identifier)))
  |-------
{1}   (no ((((d1_1 . AdstBinding) + newBinding_1) . Identifier) & ((d1_1 . BdstBinding) . Identifier)))

Rule? (dps-case "no (d1_1 . AdstBinding)")
Translating the formula.Formula translated.
Introducing case...,
this yields  2 subgoals:
ABindingPreservesStructure.2.1 :

{-1}  (no (d1_1 . AdstBinding))
[-2]  (no (((d1_1 . AdstBinding) . Identifier) & ((d1_1 . BdstBinding) . Identifier)))
[-3]  (no (((d1_1 . dstBinding) . Identifier) & (newBinding_1 . Identifier)))
  |-------
[1]   (no ((((d1_1 . AdstBinding) + newBinding_1) . Identifier) & ((d1_1 . BdstBinding) . Identifier)))

Rule? (dps-validate-goal :for 5)
Trying to validate the goal with models of size 5.
No counter-example found in that scope. The goal may be valid.
(There are available suggestions. Use M-x show-suggestions to see them.)

Rule? (postpone)
Postponing ABindingPreservesStructure.2.1.

ABindingPreservesStructure.2.2 :

[-1]  (no (((d1_1 . AdstBinding) . Identifier) & ((d1_1 . BdstBinding) . Identifier)))
[-2]  (no (((d1_1 . dstBinding) . Identifier) & (newBinding_1 . Identifier)))
  |-------
{1}   (no (d1_1 . AdstBinding))
[2]   (no ((((d1_1 . AdstBinding) + newBinding_1) . Identifier) & ((d1_1 . BdstBinding) . Identifier)))

Rule? (dps-validate-goal :for 5)
Trying to validate the goal with models of size 5.
No counter-example found in that scope. The goal may be valid.
(There are available suggestions. Use M-x show-suggestions to see them.)
No change on: (dps-validate-goal :for 5)
ABindingPreservesStructure.2.2 :

Rule? (hide 2)
Hiding formulas:  2,
this simplifies to:
ABindingPreservesStructure.2.2 :

[-1]  (no (((d1_1 . AdstBinding) . Identifier) & ((d1_1 . BdstBinding) . Identifier)))
[-2]  (no (((d1_1 . dstBinding) . Identifier) & (newBinding_1 . Identifier)))
  |-------
[1]   (no (d1_1 . AdstBinding))

Rule? (dps-validate-goal :for 5)
Trying to validate the goal with models of size 5.
Counterexample found. The goal is invalid.
```

Fig. 3. A proof fragment where an overly restrictive hypothesis is introduced

4 Reducing the Proof Search Space Using UnSAT-Cores

In this section we will discuss two techniques to reduce the proof search space during the theorem proving process. The first technique uses an iterative procedure in order to remove formulas from sequents. The second technique uses an UnSAT-core in order to determine which formulas can be removed. In Section 4.1 we discuss the iterative technique. In Section 4.2 we present the technique based on UnSAT-core extraction and compare it with the iterative technique. Finally, in Section 4.3 we present some experimental results about the usefulness of the proposed techniques.

Along this section we will assume we are willing to prove a sequent $\Gamma \vdash \Delta$ (where $\Gamma = \{ \gamma_1, \ldots, \gamma_k \}$ and $\Delta = \{ \delta_1, \ldots, \delta_m \}$) from a theory Ω containing axioms $\omega_1, \ldots, \omega_n$. In order to reduce the proof search space we will try to remove formulas from Γ, Δ and Ω. Notice that having fewer formulas actually reduces the proof search space. Many proof steps that could depend on the removed formulas (rules for instantiation, rewriting, or applying strategies) are now avoided. This reduces the number of instantiations of inference rules that the theorem prover has to consider, as well as helps the user stay focused on the relevant parts of the sequent.

4.1 An Iterative Technique to Reduce the Proof Search Space

The algorithm in Fig. 4 allows us to determine a set of formulas candidate to be removed. The algorithm attempts to remove each formula φ, and analyzes (using the Alloy Analyzer) whether the sequent obtained *after* formula φ has been removed is valid or not. If the sequent is valid, then φ can be (safely?) removed.

```
algorithm iterative_remove(Gamma, Delta, Omega)
// let Gamma = {g1,...,gk},
// let Delta = {d1,...,dm},
// let Omega = {o1,...,on}.
for i=1 to k do
   if proves(Gamma - gi, Delta, Omega) then
      Gamma = Gamma - gi
   fi
od
for i=1 to m do
   if proves(Gamma, Delta - di, Omega) then
      Delta = Delta - di
   fi
od
for i=1 to n do
   if proves(Gamma, Delta, Omega - oi) then
      Omega = Omega - oi
   fi
od
```

Fig. 4. The iterative algorithm

Procedure "proves(A, B, C)" (for $\mathtt{A} = \{ a_1, \ldots, a_{k_1} \}$, $\mathtt{B} = \{ b_1, \ldots, b_{m_1} \}$ and $\mathtt{C} = \{ c_1, \ldots, c_{n_1} \}$) checks, using the Alloy Analyzer, whether sequent $\mathtt{A} \vdash \mathtt{B}$ holds in theory C. In Alloy terms, this amounts to checking, having as facts formulas $c_1, \ldots, c_{n_1}$, the assertion

$$\left(\bigwedge_{1 \le i \le k_1} a_i \right) \Rightarrow \left(\bigvee_{1 \le j \le m_1} b_j \right) . \qquad (1)$$

Procedure proves returns true whenever the Alloy analysis does not produce a counterexample.

The previous Alloy analysis requires providing a scope for data domains. Therefore, it might be the case that the analysis of formula (1) does not return a counterexample, yet the formula indeed has counterexamples in larger scopes. This shows that this technique is not complete, since a necessary formula might be removed (this explains the question mark on "safely" above) and a valid sequent may no longer be derivable. This is not a problem in itself. Hiding formulas based on the user's intuition is not complete either. Since removing formulas does not allow us to prove previously underivable sequents, refining sequents and theories as explained is a sound rule. In Section 4.3 we will discuss experimental results in order to determine the utility of the technique.

4.2 Using the UnSAT-Core Extraction Feature to Remove Formulas

Some SAT-solvers, such as MiniSat [6] among the ones provided by the Alloy Analyzer, allow one to obtain upon completion of the analysis of an inconsistent propositional theory, an UnSAT-core. An UnSAT-core is a subset of clauses from the original inconsistent theory that is also inconsistent. The UnSAT-core extraction algorithm implemented in MiniSat produces many times small UnSAT-cores. The Alloy Analyzer converts the propositional UnSAT-core into an Alloy UnSAT-core [16] (i.e., a subset of the Alloy model that is also inconsistent if the source model was inconsistent). Notice that the algorithm in Fig. 4 actually computes an Alloy UnSAT-core. Moreover, it computes a *minimal* Alloy UnSAT-core.

Our proposal in order to remove unnecessary formulas when proving a sequent $\Gamma \vdash \Delta$ in a theory Ω (where $\Gamma = \{\gamma_1, \ldots, \gamma_k\}$, $\Delta = \{\delta_1, \ldots, \delta_m\}$ and $\Omega = \{\omega_1, \ldots, \omega_n\}$) consists on requesting the Alloy Analyzer an UnSAT-core of the Alloy model whose set of facts is Ω, and the assertion to be checked is

$$\left(\bigwedge_{1 \leq i \leq k} \gamma_i\right) \Rightarrow \left(\bigvee_{1 \leq j \leq n} \delta_j\right).$$

Upon extraction of the UnSAT-core, the Alloy Analyzer highlights those formulas from $\Gamma \cup \Delta \cup \Omega$ that are part of the UnSAT-core. We propose (as a strategy to use in the proving process) to remove those formulas that are not highlighted. Figure 5 shows a sequent from the running case-study where some of the formulas are highlighted (those that are part of the UnSAT-core). Notice that upon application of rule `dps-hide` (our new proof rule that allows to hide non-highlighted formulas) the formulas that were not highlighted are hidden. In this example (an actual sequent from the case-study) only 5 out of 23 formulas are kept in the sequent upon application of rule `dps-hide`. As with the iterative technique, addition of rule `dps-hide` makes the logic incomplete, but still sound.

Since the technique presented in Section 4.1 ends up computing an Alloy UnSat-core, a comparison to the technique presented in this section is mandatory. Notice first that the iterative technique guarantees a minimal Alloy UnSAT-core, while the UnSAT-core extraction presented in this section does not guarantee minimality. This implies that use of the UnSAT-core extraction feature provided by the Alloy Analyzer may include formulas that could be removed. An essential aspect that moved us towards adopting the technique based on UnSAT-cores is the overhead imposed on the theorem proving process. Given a sequent $\Gamma \vdash \Delta$ to be proved in a theory Ω such that $|\Gamma| = k$, $|\Delta| = m$ and $|\Omega| = n$, the iterative

```
[-1]  NonloopingDomain[d1]
[-2]  no (d1.AdstBinding.Identifier & d1.BdstBinding.Identifier)
[-3]  no (d1.AdstBinding.Identifier & d1.routing.Agent)
[-4]  no (d1.BdstBinding.Identifier & d1.routing.Agent)
[-5]  all i : Identifier | lone (i.(d1.BdstBinding))
[-6]  all i : Identifier | lone (i.(d1.routing))
[-7]  all i : Identifier | lone ((d1.BdstBinding).i)
[-8]  d1.srcBinding = ~ d1.BdstBinding
[-9]  no (Identifier.(d1.BdstBinding) & d1.AdstBinding.Identifier)
[-10] no (d1.routing.Agent & newBinding.Identifier)
[-11] no (d1.dstBinding.Identifier & newBinding.Identifier)
[-12] no (Identifier.(d1.dstBinding) & newBinding.Identifier)
[-13] no (Identifier.(d1.BdstBinding) & newBinding.Identifier)
[-14] no (Identifier.newBinding & newBinding.Identifier)
[-15] all i : Identifier | i in newBinding.Identifier =>
          ((i in Address => i in d.space) && (i in AddressPair => i.addr in d.space))
[-16] d2.endpoints = d1.endpoints
[-17] d2.space = d1.space
[-18] d2.routing = d1.routing
[-19] d2.dstBinding = d1.dstBinding + newBinding
[-20] d2.AdstBinding = d1.AdstBinding + newBinding
[-21] d2.BdstBinding = d1.BdstBinding
[-22] d2.srcBinding = d1.srcBinding
  |-------
{1}   no (d2.AdstBinding.Identifier & d2.BdstBinding.Identifier)
```

Fig. 5. A sequent with a highlighted UnSAT-core

algorithm requires $k+m+n$ calls to the SAT-solver. On the other hand, obtaining the UnSAT-core requires a single call to the SAT-solver. Ideally, the (sequent and theory) refinement process must be applied at each proof step where the sequent under analysis has new or fewer formulas. In Section 4.3 we will show experimental data supporting the election of this technique.

4.3 Experimental Results

In this section we present some experimental results we have obtained while applying the techniques presented in Sections 4.1 and 4.2. We begin by presenting some statistics about the model being verified. We have verified the model in three different ways, namely:

– Without using any technique for refining the sequents and theories. This corresponds to verification using Dynamite 1.0, as described in [7] (noted as `NoRec` – for *no recommendation* – in Table 1).
– Using the iterative algorithm in order to refine sequents (see Section 4.1). In Table 1 we note this technique as `IterRec` (for *iterative recommendation*).
– Using the UnSAT-core extraction technique presented in Section 4.2. This technique will be noted in Table 1 as `UnsatRec`.

In Table 1 we measure for each technique:

– Length of proofs (measured as the number or rule applications).
– Average number of formulas per sequent.
– Sum of occurrences of formulas in proof sequents.

- At each proof step PVS must consider sentences from the current sequent as well as the sentences from the underlying theory. We then measure the average number of such formulas over the different proof steps.
- Sum (over the proof steps) of occurrences of formulas in proof sequents or from the underlying theories.
- Number of SAT-solver calls for the iterative and the UnSAT-core-based techniques.
- Number of times the UnSAT-core obtained missed a formula necessary for closing a proof branch.
- Number of times the UnSAT-core allowed us to remove formulas that were used in the original proof because of an unnecessary detour.

In order to focus on the most relevant data we are ignoring proof steps where we prove Type Check Constraints (TCCs), which in general can be proved in a direct way. Also, we only applied the techniques (either the iterative or the UnSAT-core-based) on 69 proof steps where it was considered relevant to apply the rules. Systematic application of the iterative technique (for instance each time a new proof goal was presented by PVS) would have required in the order of 25000 calls to the SAT-solver. As a general heuristic, we set the scope for all domains (in the calls to the Alloy Analyzer) to 3.

Table 1. Measures of attributes for the employed techniques (N/A = not applicable)

	`NoRec`	`IterRec`	`UnsatRec`
Proofs' length	969	597	573
Average # of formulas per sequent	5.89	6.01	6.20
Occurrences of formulas in proofs (no theories)	5706	3590	3215
Average # of formulas in sequents or theories	34.89	35.01	7.02
Occurrences of formulas in proofs (with theories)	33807	20903	4023
# SAT-solver calls	N/A	770	69
# times UnSAT-core missed formulas	N/A	N/A	1
# times UnSAT-core avoided detour	N/A	N/A	2

Notice that proofs carried out using any of the techniques for sequent and/or theory refinement are about 40% shorter than the original proof.

In the original proof, as a means to cope with sequents' complexity, formulas that were presumed unnecessary were systematically hidden. While the average number of formulas per sequent is smaller for the original proof, having half the proof steps shows that the automated techniques are better focused on the more complex parts of proofs. This is supported by the analysis of the total number of formulas that occur in sequents. The UnSAT-core-based technique uses 56% of the formulas used in the original proof, while the iterative technique uses 63% of the formulas.

Since the underlying theory in the case-study has 29 formulas, the overhead in applying the iterative technique to formulas in the theory was too high. Therefore, the iterative technique was only applied to formulas occurring in the sequents being verified along a proof (we believe this will be the case most times). On the other hand, the UnSAT-core extraction receives the current sequent plus the underlying theory, and automatically refines *also* the theory. This explains the big difference between the average number of formulas involved in proofs (both in sequents and in the supporting theory) using the iterative technique and the UnSAT-core-based technique. Notice that this implies that in each proof step PVS had to consider significantly fewer formulas in order to suggest further proof steps.

Since proofs are shorter and each sequent contains possibly fewer formulas, the total number of formulas occurring in proofs using UnSAT-cores reduces from the original proofs in about 88% (recall that hiding was also used in the original proofs but not in an automated way, and that formulas from the underlying theory were not hidden). For the iterative technique, the number of formulas reduces in about 40%.

While using UnSAT-cores required only 69 calls to the SAT-solver, the corresponding proof steps using the iterative algorithm required 770 calls to the SAT-solver (without making calls for formulas occurring in the underlying theory). Thus, the UnSAT-core-based technique requires under 10% of the calls required by the iterative technique.

Often during the original proof necessary formulas were incorrectly hidden. We do not have precise records of the number of times this happened because those erroneous proof steps (which at the time were not considered important) were most times undone. We only kept track of 9 cases where the `reveal` command was used in order to exhibit a previously hidden formula, but these were just a few of the cases. It is worth comparing with the single case where the UnSAT-core-based technique missed a formula. This missed formula is recovered if instead of using a scope of 3 in calls to the Alloy Analyzer, scope 5 is used.

Recalling that we have proved 5 Alloy assertions, the ones corresponding to assertions `BindingPreservesDeterminism` and `BindingPreservesNonLooping` required fewer formulas during the proof based on UnSAT-cores. This shows that the original proof used unnecessary formulas that were removed using rule `dps-hide`.

A more qualitative analysis of the techniques allows us to conclude that refining sequents and theories using UnSAT-cores leads to a shift in the way the user faces the proving process. Looking at the (usually few) remaining formulas after `dps-hide` is applied helped the user gain a better understanding on the property to be proved.

5 Related Work

In this section we discuss work related to Dynamite on the combination of SAT-solving with theorem proving, and more specific work on the applications of UnSAT-cores. Using SAT-solving in the context of first-order theorem proving is not new. The closest works to Dynamite 1.0 are the thesis [20] and the article

[5]. They follow the idea of our 2007 article [7] of using a model generator to look for counterexamples of formulas being proved by a theorem prover. Previous articles such as [19] only focus on using the SAT-solver to prove propositional tautologies and use the resolution proofs provided by the SAT-solver to guide the theorem prover proofs. This is more restricted than Dynamite 1.0 in that Dynamite is not constrained to propositional formulas. The 2009 article [4] introduces Nitpick, which is based (as Dynamite 1.0) on Kodkod [17]. Nitpick, as Dynamite 1.0, helps during the theorem proving process by detecting that a non-theorem is being proved. It is worth emphasizing that none of these articles make use of UnSAT-cores during the proving process. Reducing the number of sentences in sequents has been acknowledged as an important problem by the Automated Theorem Proving community. The tool MaLARea [18] reduces sets of hypotheses using machine learning techniques. Sledgehammer [3], uses automated theorem provers to select axioms during interactive theorem proving. The iterative technique presented in Section 4.1 shows resemblance with [13], but [13] uses the Darwin model finder tool to convert first-order sentences into function-free clause sets. No notion of UnSAT-cores is provided or used. The SRASS system [15] uses the ideas presented in [13] and complements them with a notion of syntactic relevance, but does not make use of UnSAT-cores. Last, theorem proving of Alloy assertions was first considered in [2]. The theorem prover Prioni translated Alloy sentences to first-order logic sentences, and the theorem prover Athena [1] was used on the resulting formula. Notice that the translation removes the relational flavor of Alloy, and therefore Alloy users are confronted with an unfamiliar formalism. While Prioni is a theorem prover for the Alloy language, it does not make use of the Alloy Analyzer to contribute to the proving process.

6 Conclusions and Further Work

In this article we have presented two techniques for the elimination of superfluous formulas in sequents and theories. The iterative technique allows us to remove formulas but is not appropriate in the context of sequents or theories containing many formulas because it requires many calls to the SAT-solver. It is appropriate if we restrict the application of the technique to formulas occurring in sequents and forget about formulas in the supporting theories. To the best of our knowledge, the idea of refining sequents and theories using UnSAT-cores is novel and shows (on the experiments reported) to contribute to produce shorter and more focused proofs.

This article is part of a more ambitious project on using the unsatisfiability proofs produced by the SAT-solver in order to suggest proof steps, but making special emphasis on proof steps that use quantifier-related proof rules. We plan to continue working in this direction. The current DPS 2.0 interface is based on EMACS. We are developing a new interface that shows closer resemblance to the Alloy Analyzer's interface (including exhibiting counterexamples using the graphic capabilities provided by the Alloy Analyzer).

References

1. Arkoudas, K.: Type-ω DPLs, MIT AI Memo 2001-27 (2001)
2. Arkoudas, K., Khurshid, S., Marinov, D., Rinard, M.: Integrating Model Checking and Theorem Proving for Relational Reasoning. In: Proceedings of RelMiCS 2003, Springer, Heidelberg (2003)
3. Böhme, S., Nipkow, T.: Sledgehammer: Judgement Day. In: IJCAR 2010 (to appear, 2010)
4. Blanchette, J.C., Nipkow, T.: Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder. In: TAP 2009 (2009)
5. Dunets, A., Schellhorn, G., Reif, W.: Automated Flaw Detection in Algebraic Specifications. Journal of Automated Reasoning (2010)
6. Eén, N., Sörensson, N.: MiniSat-p-v1.14. A proof-logging version of MiniSat (September 2006)
7. Frias, M.F., López Pombo, C.G., Moscato, M.M.: Alloy Analyzer+PVS in the Analysis and Verification of Alloy Specifications. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 587–601. Springer, Heidelberg (2007)
8. Hoare, C.A.R., Leavens, G.T., Misra, J., Shankar, N.: The Verified Software Initiative: A Manifesto (2007)
9. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology 11, 256–290 (2002)
10. Jackson, D., Schechter, I., Shlyakhter, I.: Alcoa: the Alloy Constraint Analyzer. In: ICSE 2000, pp. 730–733 (2000)
11. Kang, E., Jackson, D.: Formal Modeling and Analysis of a Flash Filesystem in Alloy. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 294–308. Springer, Heidelberg (2008)
12. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS (LNAI), vol. 607, pp. 148–752. Springer, Heidelberg (1992)
13. Pudlák, P.: Semantic Selection of Premisses for Automated Theorem Proving. In: Proceedings of ESARLT 2007, pp. 27–44 (2007)
14. Ramananandro, T.: Mondex, an electronic purse: specification and refinement checks with the Alloy model-finding method. Formal Aspects of Computing 20(1), 21–39 (2008)
15. Sutcliffe, G., Puzis, Y.: SRASS a semantic relevance axiom selection system (2007), http://www.cs.miami.edu/~tptp/ATPSystems/SRASS/
16. Torlak, E., Chang, F., Jackson, D.: Finding Minimal Unsatisfiable Cores of Declarative Specifications. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 326–341. Springer, Heidelberg (2008)
17. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
18. Urban, J.: MaLARea: a Metasystem for Automated Reasoning in Large Theories. In: Proceedings of ESARLT 2007, pp. 45–58 (2007)
19. Weber, T.: Integrating a SAT Solver with an LCF-style Theorem Prover. In: Proceedings of PDPAR 2005. ENTCS, vol. 144(2), pp. 67–78.
20. Weber, T.: SAT-based Finite Model Generation for Higher-Order Logic, Ph.D. Thesis, TUM (2008)
21. Zave, P.: Compositional binding in network domains. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 332–347. Springer, Heidelberg (2006)

Complete Calculi for Structured Specifications in Fork Algebra

Carlos Gustavo Lopez Pombo[1,3] and Marcelo Fabiùn Frias[2,3]

[1] Department of computer science, Facultad de ciencias exactas y naturales, Universidad de Buenos Aires
[2] Department of Computer Engineering, Buenos Aires Institute of Technology (ITBA)
[3] CONICET
clpombo@dc.uba.ar, mfrias@itba.edu.ar

Abstract. In previous articles we presented $\mathbf{Ar}_{\mathbf{g}}$*entum*, a tool for reasoning across heterogeneous specifications based on the language of fork algebras. $\mathbf{Ar}_{\mathbf{g}}$*entum*'s foundations were formalized in the framework of institutions. The formalization made simple to describe a methodology capable of producing a complete system desription from partial views, eventually written in different logical languages.

Structured specifications were introduced by Sannella and Tarlecki and extensively studied by Borzyszkowski. The latter also presented conditions under which the calculus for structured specifications is complete. Using fork algebras as a "universal" institution capable of representing expressive logics (such as dynamic and temporal logics), requires using a fork language that includes a reflexive-transitive closure operator. The calculus thus obtained does not meet the conditions required by Borzyszkowski.

In this article we present structure building operators (SBOs) over fork algebras, and provide a complete calculus for these operators.

1 Introduction and Motivation

Modeling languages such as the Unified Modeling Language (UML) [1] allow us to model a system through various diagrams. Each diagram provides a view of the system under development. This view-centric approach to software modeling has its advantages and disadvantages. Two advantages are clear: *a*) decentralization of the modeling process. Several engineers may be modeling different views of the same system simultaneously, and *b*) separation of concerns is enforced.

At the same time this modeling process evolved, several results were produced on the interpretability of logics to extensions of the theory of fork algebras [2]. An interpretation of a logic L to fork algebras consists on a mapping $T_L : Sen_L \to Sen_{FA}$ satisfying the following interpretability condition:

$$\Gamma \models_L \alpha \iff \{\, T_L(\gamma) \mid \gamma \in \Gamma \,\} \vdash_{FA} T_L(\alpha) \ .$$

So far, interpretability results have been produced for classical first-order logic with equality [2, Cap. 5], monomodal logics and propositional dynamic logic

[2, Cap. 6], first-order dynamic logic [3], propositional linear temporal logic [4] and first-order linear temporal logic [5]. Other attractive features of this class of algebras are that they are isomorphic to algebras whose domain is a set of binary relations, and that they posses a finite equational calculus [2, Cap. 4].

The idea of having heterogeneous specifications and reasoning across them is not new. A vast amount of work on the subject has been done based on Goguen and Burstall's notion of *institution* [6]. Institutions capture in an abstract way the model theory of a logic. They can be related by means of different kinds of mappings such as institution morphisms [6] and institution representations [7]. These mappings between institutions are extensively discussed by Tarlecki in [8]. Representations allow us to encode poorer institutions into a richer one. In [8], Tarlecki goes even further in presenting the way heterogeneous specifications must be manipulated when he writes:

> "... this suggests that we should strive at a development of a convenient to use proof theory (with support tools!) for a sufficiently rich "universal" institution, and then reuse it for other institutions linked to it by institution representations."

These results constitute the foundations of the $\mathbf{A}r_{\mathbf{g}}$*entum* project, presented in [9]. $\mathbf{A}r_{\mathbf{g}}$*entum* is a CASE tool aimed at the analysis of heterogeneous models of software. A system description is a collection of theory presentations coming from different logics, and analysis of the heterogeneous model is achieved by interpreting the presentations to fork algebras and analyzing the resulting fork-algebraic specification by available tool support.

An additional concern is how to deal with structured specifications. In practice, specifications are built modularly. In [10], Borzyszkowski gave a set of structure building operations (SBOs) and proved that under certain conditions structured specifications over a given logic can be translated, by means of the application of a representation map, to structured specifications over another logic. Thus, an approach that pretends to act as a foundation of a heterogeneous framework following Tarleckis approach (for example, the fork algebras) should support structured specifications. Also in [10], Borzyszkowski presented a logical system for SBOs, as well as an extensive discussion on the conditions under which that calculus is complete. One of these conditions is that the underlying institution must have a complete calculus. This is indeed the case for fork algebras. The other conditions establish requirements which unfortunately the calculus for fork algebras does not meet.

When working with heterogeneous specifications within a "universal" institution, Borzyszkowski's conditions are hard to satisfy. A logic powerful enough to interpret other logics useful in software specification must be expressive enough. Thus, it is unlikely that such a logical system could satisfy conditions such as compactness, interpolation, infinite conjunction and implications, or interesting combinations of them. In [9] we showed why fork algebras are appropriate as a "universal" institution. For example, surveying interpretability results like those for first-order dynamic logic [3], it is easy to see that any candidate to "universal" institution must have an expressive power capable of characterizing, for instance, reflexive-transitive closure, thus forcing the introduction of some kind of infinitary rule.

In this paper we present a complete calculus for SBOs over fork algebras, and discuss its scope and limitations under several conditions.

The article is organized as follows: In Sec. 2 we present fork algebras as the logical system we will use as universal institution. In Sec. 3 we provide the basic definitions such as institution, entailment system, and structure building operations. In Sec. 4 we develop the contribution of the article by analyzing the calculus proposed by Borzyszkowski and discussing its possibilities and limitations. Finally, in Sec. 5 we draw some conclusions.

2 Fork Algebras

Full proper closure fork algebras with urelements (denoted by fPCFAU) are extensions of relation algebras [11]. In order to introduce this class, we introduce first the class of *star proper closure fork algebras with urelements* (denoted by $\star$PCFAU).

Definition 1. *Let U be a nonempty set. A $\star$PCFAU is a two sorted structure $\left\langle U, 2^{U\times U}, \cup, \cap, ^{-}, \emptyset, U\times U, \circ, Id, ^{\smile}, \nabla, ^{\diamond}, ^{*}, \star \right\rangle$ such that*

- *$\star : U \times U \to U$ is one to one, but not surjective,*
- *Id is the identity relation on the set U,*
- *$\cup$, $\cap$ and $^{-}$ stand for set union, intersection and complement relative to $U \times U$, respectively,*
- *$x^{\diamond}$ is the set choice operator defined by the condition:*

$$x^{\diamond} \subseteq x \text{ and } |x^{\diamond}| = 1 \iff x \neq \emptyset,$$

- *$\circ$ is relational composition, $^{\smile}$ is transposition, and * is reflexive-transitive closure,*
- *∇, the* fork *operator, is defined by the condition:*

$$S \nabla T = \{ \langle x, y \star z \rangle \mid \langle x, y\rangle \in S \ \wedge \ \langle x, z\rangle \in T \} \ .$$

Notice that $x^{\diamond}$ denotes an arbitrary pair in x. This is why $x^{\diamond}$ is called a *choice* operator. Function $\star$ is used to encode pairs. For example, in the case of the interpretability of forst-order logic in fork algebras, $\star$ is used to group elements of the domain to represent a valuation of the variables ($a_1 \star a_2 \star ... \star a_n$ represents valuations ν satisfying $\nu(v_i) = a_i$). The fact it is not surjective implies the existence of elements that do not encode pairs. These elements, called *urelements*, will be used to represent the elements from the carriers of the translated logics.

Definition 2. *We define* fPCFAU $=$ Rd $\star$PCFAU*, where* Rd *takes reducts to structures of the form $\left\langle 2^{U\times U}, \cup, \cap, ^{-}, \emptyset, U\times U, \circ, Id, ^{\smile}, \nabla, ^{\diamond}, ^{*} \right\rangle$ (the sort U and the function $\star$ are forgotten).*

We will refer to the carrier of an algebra $\mathcal{A} \in \mathsf{fPCFAU}$ as $|\mathcal{A}|$.

The variety generated by fPCFAU (the class of *full proper closure fork algebras with urelements*) has a complete ([3, Theorem 1]) equational calculus (the ω-calculus for closure fork algebras with urelements – ω-CCFAU) to be introduced next. In order to present the calculus, we provide the grammar for formulas, the axioms of the calculus, and the proof rules. For the sake of simplifying the notation, we will denote the relation $U \times U$ by 1, and the relation $\overline{1 \nabla 1} \cap Id$ by Id_{U}. Relation Id_{U} is the subset of the identity relation that relates the urelements.

Definition 3. *Let $\mathcal{V}$ be a set of relation variables, then the set of ω-CCFAU terms is the smallest set $Term(\mathcal{V})$ satisfying:*

- $\{\emptyset, 1, Id\} \subseteq Term(\mathcal{V})$,
- *If $x, y \in Term(\mathcal{V})$, then $\{\breve{x}, x^*, x^\diamond, x \cup y, x \cap y, x \circ y, x \nabla y\} \subseteq Term(\mathcal{V})$.*

Definition 4. *Let $\mathcal{V}$ be a set of relation variables, then the set of ω-CCFAU formulas is the set of identities $t_1 = t_2$, with $t_1, t_2 \in Term(\mathcal{V})$.*

Definition 5. *The identities described in Forms. (1) – (5) are axioms[1] of ω-CCFAU.*

1. *A set of identities axiomatizing the relational calculus [11].*
2. *The following axioms for the fork operator:*

$$\begin{aligned} x \nabla y &= (x \circ (Id \nabla 1)) \cap (y \circ (1 \nabla Id)) , \\ (x \nabla y) \circ (z \nabla w)^{\smile} &= (x \circ \breve{z}) \cap (y \circ \breve{w}) , \\ (Id \nabla 1)^{\smile} \nabla (1 \nabla Id)^{\smile} &\leq Id. \end{aligned}$$

3. *The following axioms for the choice operator [12, p. 324]:*

$$x^\diamond \circ 1 \circ \breve{x^\diamond} \leq Id, \quad \breve{x^\diamond} \circ 1 \circ x^\diamond \leq Id, \quad 1 \circ (x \cap x^\diamond) \circ 1 = 1 \circ x \circ 1 \ .$$

4. *The following axioms for the Kleene star:*

$$x^* = Id \ \cup \ x \circ x^*, \quad x^* \circ y \leq y \ \cup \ x^* \circ (\overline{y} \ \cap \ x \circ y) \ .$$

5. *An axiom forcing a nonempty set of urelements.*

$$1 \circ Id_{\mathsf{U}} \circ 1 = 1 \ .$$

Definition 6. *The inference rules for the calculus ω-CCFAU are those of equational logic (see for instance [13, p. 94]), extended by adding the following inference rule[2]:*

[1] Since the calculus of relations extends the Boolean calculus, we will denote by $\leq$ the ordering induced by the Boolean calculus in ω-CCFAU. As it is usual, $x \leq y$ is a shorthand for $x \cup y = y$.

[2] Given $i > 0$, by x^i we denote the relation inductively defined as follows: $x^1 = x$, and $x^{i+1} = x \circ x^i$.

$$\frac{\vdash Id \leq y \qquad x^i \leq y \vdash x^{i+1} \leq y \quad (\forall i \in \mathbb{N})}{\vdash x^* \leq y}$$

Notice that only extralogical symbols belong to an equational or first-order signature. Symbols such as $=$ in equational logic, or $\vee$ in first-order logic, have a meaning that is univoquely determined by the carriers and the interpretation of the extralogical symbols. Similarly, once the field of a fPCFAU has been fixed, all the operators can be assigned a standard meaning. This gives rise to the following definition of fPCFAU-signature . Given $\mathfrak{A}$ a proper closure fork algebra, then $\leq$ is set inclusion.

Definition 7. *A* fPCFAU *signature is a set of function symbols $\{f_j\}_{j \in \mathcal{J}}$. Each function symbol comes equipped with its arity. Notice that since* fPCFAU*s have only one sort, the arity is a natural number.*

The set of fPCFAU signatures will be denoted as $Sign_{\mathsf{fPCFAU}}$. Actually, in order to interpret the logics mentioned in Sec. 1, constant relational symbols (rather than functions in general) suffice. Since new operators may be necessary in order to interpret new logics in the future, signatures will be allowed to contain functions of arbitrary rank.

In order to extend the definitions of terms (Def. 3) and formulas (Def. 4) to fPCFAU signatures, we need to add the following rule:

– If $t_1, \ldots, t_{ari(f_j)} \in Term(\mathcal{V})$, then $f_j(t_1, \ldots, t_{arity(f_j)}) \in Term(\mathcal{V})$ (for all $j \in \mathcal{J}$).

If $\Sigma \in Sign_{\mathsf{fPCFAU}}$, the set of Σ-terms will be denoted as $Term_\Sigma$. In the same way, Sen_Σ will denote the set of equalities between Σ-terms (i.e. the set of Σ-formulas).

Definition 8. *Let $\Sigma = \{f_j\}_{j \in \mathcal{J}} \in Sign_{\mathsf{fPCFAU}}$, then $\mathcal{M} = \left\langle \mathcal{P}, \{\underline{f_j}\}_{j \in \mathcal{J}} \right\rangle \in Mod_\Sigma$ iff $\mathcal{P} \in$* fPCFAU, *and $\underline{f_j} : |\mathcal{P}|^{arity(f_j)} \to |\mathcal{P}|$, for all $j \in \mathcal{J}$. Then, we denote by $m_\mathcal{M} : Term_\Sigma \to |\mathcal{P}|$ the function that interprets terms in model $\mathcal{M}$.*

Definition 9. *Let $\Sigma \in Sign_{\mathsf{fPCFAU}}$, then $\models^\Sigma_{\mathsf{fPCFAU}} \subseteq Mod_\Sigma \times Sen_\Sigma$ is defined as follows: $\mathcal{M} \models^\Sigma_{\mathsf{fPCFAU}} t_1 = t_2$ iff $m_\mathcal{M}(t_1) = m_\mathcal{M}(t_2)$.*

3 Institutions and Structured Specifications

The theory of institutions was presented by Goguen and Burstall in [6]. Institutions provide a formal and generic definition of what a logical system is, and of how specifications in a logical system can be structured [14]. Institutions have evolved in a number of directions, from an abstract theory of software specification and development [15] to a very general version of abstract model theory [16], and offered a suitable formal framework for addressing heterogeneity [17,18], including applications to the UML [19]. The following definitions were taken from [7].

Definition 10. [Institution]
An institution *is a structure of the form* $\langle \mathsf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \{\models^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|}\rangle$ *satisfying the following conditions:*

- Sign *is a category of signatures,*
- $\mathbf{Sen} : \mathsf{Sign} \to \mathsf{Set}$ *is a functor (let* $\Sigma \in |\mathsf{Sign}|$*, then* $\mathbf{Sen}(\Sigma)$ *returns the set of* Σ*-sentences),*
- $\mathbf{Mod} : \mathsf{Sign}^{\mathsf{op}} \to \mathsf{Cat}$ *is a functor (let* $\Sigma \in |\mathsf{Sign}|$*, then* $\mathbf{Mod}(\Sigma)$ *returns the category of* Σ*-models),*
- $\{\models^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|}$*, where* $\models^{\Sigma} \subseteq |\mathbf{Mod}(\Sigma)| \times \mathbf{Sen}(\Sigma)$*, is a family of binary relations,*

and for any signature morphism $\sigma : \Sigma \to \Sigma'$*,* Σ*-sentence* $\phi \in \mathbf{Sen}(\Sigma)$ *and* Σ'*-model* $\mathcal{M}' \in |\mathbf{Mod}(\Sigma)|$ *the following* $\models$*-invariance condition holds:*

$$\mathcal{M}' \models^{\Sigma'} \mathbf{Sen}(\sigma)(\phi) \ \textit{ iff } \ \mathbf{Mod}(\sigma^{\mathsf{op}})(\mathcal{M}') \models^{\Sigma} \phi \, .$$

Let $\Sigma \in |\mathsf{Sign}|$ and $\Gamma \subseteq \mathbf{Sen}(\Sigma)$, then we define the functor $\mathbf{Mod}(\Sigma, \Gamma)$ as the full subcategory of $\mathbf{Mod}(\Sigma)$ determined by those models $\mathcal{M} \in |\mathbf{Mod}(\Sigma)|$ such that for all $\gamma \in \Gamma$, $\mathcal{M} \models^{\Sigma} \gamma$. In addition, it is possible to define a relation $\models^{\Sigma}$ between sets of formulas and formulas in the following way: let $\alpha \in \mathbf{Sen}(\Sigma)$, then:

$$\Gamma \models^{\Sigma} \alpha \ \text{ if and only if } \ \mathcal{M} \models^{\Sigma} \alpha \ \text{ for all } \mathcal{M} \in |\mathbf{Mod}(\Sigma, \Gamma)|.$$

Definition 11. [Entailment system]
An entailment system *is a structure of the form* $\langle \mathsf{Sign}, \mathbf{Sen}, \{\vdash^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|}\rangle$ *satisfying the following conditions:*

- Sign *is a category of signatures,*
- $\mathbf{Sen} : \mathsf{Sign} \to \mathsf{Set}$ *is a functor (let* $\Sigma \in |\mathsf{Sign}|$*, then* $\mathbf{Sen}(\Sigma)$ *returns the set of* Σ*-sentences),*
- $\{\vdash^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|}$*, where* $\vdash^{\Sigma} \subseteq 2^{\mathbf{Sen}(\Sigma)} \times \mathbf{Sen}(\Sigma)$*, is a family of binary relations such that for any* $\Sigma, \Sigma' \in |\mathsf{Sign}|$*,* $\{\phi\} \cup \{\phi_i\}_{i \in \mathcal{I}} \subseteq \mathbf{Sen}(\Sigma)$*,* $\Gamma, \Gamma' \subseteq \mathbf{Sen}(\Sigma)$ *the following conditions are satisfied:*
  1. *reflexivity:* $\{\phi\} \vdash^{\Sigma} \phi$*,*
  2. *monotonicity: if* $\Gamma \vdash^{\Sigma} \phi$ *and* $\Gamma \subseteq \Gamma'$*, then* $\Gamma' \vdash^{\Sigma} \phi$*,*
  3. *transitivity: if* $\Gamma \vdash^{\Sigma} \phi_i$ *for all* $i \in \mathcal{I}$ *and* $\{\phi_i\}_{i \in \mathcal{I}} \vdash^{\Sigma} \phi$*, then* $\Gamma \vdash^{\Sigma} \phi$*, and*
  4. $\vdash$*-translation: if* $\Gamma \vdash^{\Sigma} \phi$*, then for any morphism* $\sigma : \Sigma \to \Sigma'$ *in* Sign*,* $\mathbf{Sen}(\sigma)(\Gamma) \vdash^{\Sigma'} \mathbf{Sen}(\sigma)(\phi)$*.*

Definition 12. *Let* $\langle \mathsf{Sign}, \mathbf{Sen}, \{\vdash^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|}\rangle$ *be an entailment system, then* Th*, its category of theories, is a pair* $\langle \mathcal{O}, \mathcal{A}\rangle$ *such that:*

- $\mathcal{O} = \{\, \langle \Sigma, \Gamma\rangle \mid \Sigma \in |\mathsf{Sign}| \textit{ and } \Gamma \subseteq \mathbf{Sen}(\Sigma) \,\}$*, and*
- $\mathcal{A} = \left\{ \sigma : \langle \Sigma, \Gamma\rangle \to \langle \Sigma', \Gamma'\rangle \;\middle|\; \begin{array}{l} \langle \Sigma, \Gamma\rangle, \langle \Sigma', \Gamma'\rangle \in \mathcal{O}, \\ \sigma : \Sigma \to \Sigma' \textit{ is a morphism in } \mathsf{Sign} \textit{ and} \\ \textit{for all } \gamma \in \Gamma, \Gamma' \vdash^{\Sigma'} \mathbf{Sen}(\sigma)(\gamma) \end{array} \right\}.$

In addition, if a morphism $\sigma : \langle \Sigma, \Gamma \rangle \to \langle \Sigma', \Gamma' \rangle$ satisfies $\mathbf{Sen}(\sigma)(\Gamma) \subseteq \Gamma'$ it is called *axiom preserving*. This defines the category Th_0 by keeping only those morphisms of Th that are axiom preserving. It is easy to notice that Th_0 is a complete subcategory of Th. Now, if we consider the definition of $\mathbf{Mod}$, extended to signatures and set of sentences, we get a functor $\mathbf{Mod} : \mathsf{Th}^{\mathsf{op}} \to \mathsf{Cat}$ defined as follows: let $T = \langle \Sigma, \Gamma \rangle \in |\mathsf{Th}|$, then $\mathbf{Mod}(T) = \mathbf{Mod}(\Sigma, \Gamma)$.

Definition 13. [Logic]
A logic *is a structure of the form* $\langle \mathsf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \{\vdash^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|}, \{\models^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|} \rangle$ *satisfying the following conditions:*

– $\langle \mathsf{Sign}, \mathbf{Sen}, \{\vdash^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|} \rangle$ *is an entailment system,*
– $\langle \mathsf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \{\models^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|} \rangle$ *is an institution, and*
– *the following* soundness *condition is satisfied: for any* $\Sigma \in |\mathsf{Sign}|$, $\phi \in \mathbf{Sen}(\Sigma)$, $\Gamma \subseteq \mathbf{Sen}(\Sigma)$, $\Gamma \vdash^{\Sigma} \phi \Longrightarrow \Gamma \models^{\Sigma} \phi$.

A logic is complete *if in addition the following condition is also satisfied: for any* $\Sigma \in |\mathsf{Sign}|$, $\phi \in \mathbf{Sen}(\Sigma)$, $\Gamma \subseteq \mathbf{Sen}(\Sigma)$, $\Gamma \vdash^{\Sigma} \phi \Longleftarrow \Gamma \models^{\Sigma} \phi$.

Next we provide some definitions that will be useful in further sections.

Definition 14. [Interpolation and weak interpolation]
An institution $\langle \mathsf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \{\models^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|} \rangle$ *has the* interpolation property *if and only if for any* $t'_1 : \Sigma_1 \to \Sigma', t'_2 : \Sigma_2 \to \Sigma'$ *pushout in* Sign *for* $t_1 : \Sigma \to \Sigma_1, t_2 : \Sigma \to \Sigma_2$, *and* $\varphi_i \in \mathbf{Sen}(\Sigma_i)$ *for* $i = 1, 2$, *if* $\mathbf{Sen}(t'_1)(\varphi_1) \models^{\Sigma'} \mathbf{Sen}(t'_2)(\varphi_2)$, *then there exists* $\varphi \in \mathbf{Sen}(\Sigma)$ *(called the interpolant of* φ_1 *and* φ_2*) such that* $\varphi_1 \models^{\Sigma_1} \mathbf{Sen}(t_1)(\varphi)$ *and* $\mathbf{Sen}(t_2)(\varphi) \models^{\Sigma_2} \varphi_2$.

In a similar way, it is said to have the weak interpolation property *if and only if for any* $t'_1 : \Sigma_1 \to \Sigma', t'_2 : \Sigma_2 \to \Sigma'$ *pushout in* Sign *for* $t_1 : \Sigma \to \Sigma_1, t_2 : \Sigma \to \Sigma_2$, *and* $\varphi_i \in \mathbf{Sen}(\Sigma_i)$ *for* $i = 1, 2$, *if* $\mathbf{Sen}(t'_1)(\varphi_1) \models^{\Sigma'} \mathbf{Sen}(t'_2)(\varphi_2)$, *then there exists* $\Gamma \subseteq \mathbf{Sen}(\Sigma)$ *(called the interpolant of* φ_1 *and* φ_2*) such that* $\varphi_1 \models^{\Sigma_1} \mathbf{Sen}(t_1)(\Gamma)$ *and* $\mathbf{Sen}(t_2)(\Gamma) \models^{\Sigma_2} \varphi_2$.

Definition 15. [Weak amalgamation]
An institution $\langle \mathsf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \{\models^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|} \rangle$ *has the* weak amalgamation property *if and only if for any* $t'_1 : \Sigma_1 \to \Sigma', t'_2 : \Sigma_2 \to \Sigma'$ *pushout in* Sign *for* $t_1 : \Sigma \to \Sigma_1, t_2 : \Sigma \to \Sigma_2$, *and for any models* $\mathcal{M}_1 \in |\mathbf{Mod}(\Sigma_1)|$ *and* $\mathcal{M}_2 \in |\mathbf{Mod}(\Sigma_2)|$ *such that* $\mathbf{Mod}(t_1)(\mathcal{M}_1) = \mathbf{Mod}(t_2)(\mathcal{M}_2)$, *then there exists* $\mathcal{M}' \in |\mathbf{Mod}(\Sigma')|$ *such that* $\mathbf{Mod}(t'_1)(\mathcal{M}') = \mathcal{M}_1$ *and* $\mathbf{Mod}(t'_2)(\mathcal{M}') = \mathcal{M}_2$.

From now on we will work with specifications as they were defined in [10]. Given an institution $\mathbb{I} = \langle \mathsf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \{\models^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|} \rangle$, for any specification SP over $\mathbb{I}$ we will denote its signature as $\mathbf{Sig}[SP] \in |\mathsf{Sign}|$, and its class of models as $\mathbf{Mod}[SP] \subseteq \mathbf{Mod}(\mathbf{Sig}[SP])$[3]. If $\mathbf{Sig}[SP] = \Sigma$, then we will call SP a Σ-specification, and we will denote the class of Σ-specifications as Spec_{Σ}.

[3] Notice that there are two operators **Mod**, the first one applying to structured specifications and the second one being the model functor of the institution.

Definition 16 [Structure building operations]. *The class of specifications over an institution* $\langle \mathsf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \{\models^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|}\rangle$ *are defined as follows*

- *Any pair* $\langle \Sigma, \Gamma \rangle$*, where* $\Sigma \in |\mathsf{Sign}|$ *and* $\Gamma \subseteq \mathbf{Sen}(\Sigma)$*, is a specification, also called* flat specification *or* presentation*, such that:*
  $\mathbf{Sig}[\langle \Sigma, \Gamma \rangle] = \Sigma$, $\mathbf{Mod}[\langle \Sigma, \Gamma \rangle] = |\mathbf{Mod}(\langle \Sigma, \Gamma \rangle)|$
- *Let* $\Sigma \in |\mathsf{Sign}|$*, then given* $SP_1, SP_2 \in \mathsf{Spec}_{\Sigma}$*,* $SP_1 \cup SP_2$ *is a* Σ*-specification such that:*
  $\mathbf{Sig}[SP_1 \cup SP_2] = \Sigma$, $\mathbf{Mod}[SP_1 \cup SP_2] = \mathbf{Mod}[SP_1] \cap \mathbf{Mod}[SP_2]$
- *Let* $\Sigma, \Sigma' \in |\mathsf{Sign}|$*, then given* $SP \in \mathsf{Spec}_{\Sigma}$ *and a morphism* $\sigma : \Sigma \to \Sigma'$*, then* **translate** SP **by** σ *is a* Σ'*-specification such that:*
  $\mathbf{Sig}[\textbf{translate } SP \textbf{ by } \sigma] = \Sigma'$,
  $\mathbf{Mod}[\textbf{translate } SP \textbf{ by } \sigma] = \{\, \mathcal{M}' \mid \mathbf{Mod}(\sigma)(\mathcal{M}') \in \mathbf{Mod}[SP] \,\}$
- *Let* $\Sigma, \Sigma' \in |\mathsf{Sign}|$*, then given* $SP' \in \mathsf{Spec}_{\Sigma'}$ *and a morphism* $\sigma : \Sigma \to \Sigma'$*, then* **derive from** SP' **by** σ *is a* Σ*-specification such that:*
  $\mathbf{Sig}[\textbf{derive from } SP' \textbf{ by } \sigma] = \Sigma$,
  $\mathbf{Mod}[\textbf{derive from } SP' \textbf{ by } \sigma] = \{\, \mathbf{Mod}(\sigma)(\mathcal{M}') \mid \mathcal{M}' \in \mathbf{Mod}[SP'] \,\}$

The operations introduced in the previous definition are referred as structure building operations or SBOs, and express a mechanism to put specifications together in a structured way. The operators **Sig** and **Mod** help us retrieve both the signature and the corresponding class of models for a given structured specification.

Definition 17. *Given* SP_1 *and* SP_2 *specifications, we say that* SP_1 *is equivalent to* SP_2 *(denoted* $SP_1 \equiv SP_2$*) if and only if* $\mathbf{Sig}[SP_1] = \mathbf{Sig}[SP_2]$ *and* $\mathbf{Mod}[SP_1] = \mathbf{Mod}[SP_2]$*.*

Definition 18. *Given* SP *a* Σ*-specification, and* $\alpha \in |\mathbf{Sen}(\Sigma)|$*.* α *is a semantic consequence of* SP *(denoted* $SP \models_{\Sigma} \alpha$*) if and only if* $\mathbf{Mod}[SP] \models^{\Sigma} \alpha$*.*[4]

As it was presented in [10] structured specifications have a normal form of the shape **derive from** SP' **by** t, where SP' is a flat specification. This normal form is obtained by the application of the operator **nf** [10, Def. 3.7][5]. Also in [10] the following theorem is proved.

Theorem 1. *Let* SP *be a* Σ*-specification over the institution* $\mathbb{I}$*, if* $\mathbb{I}$ *has the weak amalgamation property, then* $\mathbf{nf}(SP) \equiv SP$*.*

[4] As $\mathbf{Mod}[SP]$ is a class of models, we define $\mathbf{Mod}[SP] \models^{\Sigma} \alpha$ if and only if for all $\mathcal{M} \in \mathbf{Mod}[SP]$, $\mathcal{M} \models^{\Sigma} \alpha$. Also notice the difference between $\models_{\Sigma}$ and $\models^{\Sigma}$, the first one being the satisfaction relation between structured specifications and formulas, and the second one being the satisfaction relation of the underlying institution.

[5] The intuition behind the operator **nf** is that it flattens the specification by translating the axioms to the "richest" signature using the pushouts in Sign followed by the derivation of the resulting flat specification to a signature having the symbols that must remain visible.

4 Complete Calculi for Structured Specifications in Fork Algebras

In [10] Borzyszkowski presented a calculus for structured specifications and gave sufficient conditions under which it is complete. In this section we will explore conditions under which this calculus is complete when specifications are structured over Fork Algebras.

4.1 Scope and Limitations of Borzyszkowski's Calculus for Structured Specifications

We start by reviewing the scope and limitations of Borzyszkowski's calculus for structured specifications, by analyzing the conditions presented in [10] when they are instantiated for the logic of full proper closure fork algebras with urelements.

Definition 19. *Let* $\mathbb{I} = \langle \mathsf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \{\vdash^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|}, \{\models^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|} \rangle$ *be the logic of full proper closure fork algebras with urelements. Then, the following rules, define a* Sign*-indexed family of entailment relations.*

$$\frac{\{SP \vdash_{\Sigma} \psi\}_{\psi \in \Delta} \qquad \Delta \vdash^{\Sigma} \varphi}{SP \vdash_{\Sigma} \varphi}\,\textit{[CR]} \qquad \frac{\Gamma \vdash^{\Sigma} \varphi}{\langle \Sigma, \Gamma \rangle \vdash_{\Sigma} \varphi}\,\textit{[basic]}$$

$$\frac{SP' \vdash_{\Sigma'} \mathbf{Sen}(\sigma)(\varphi)}{\textbf{derive from } SP' \textbf{ by } \sigma \vdash_{\Sigma} \varphi}\,\textit{[derive]}$$

$$\frac{SP_1 \vdash_{\Sigma} \varphi}{SP_1 \cup SP_2 \vdash_{\Sigma} \varphi}\,\textit{[sum1]} \qquad \frac{SP_2 \vdash_{\Sigma} \varphi}{SP_1 \cup SP_2 \vdash_{\Sigma} \varphi}\,\textit{[sum2]}$$

$$\frac{SP \vdash_{\Sigma} \varphi}{\textbf{translate } SP \textbf{ by } \sigma \vdash_{\Sigma'} \mathbf{Sen}(\sigma)(\varphi)}\,\textit{[translate]}$$

Definition 20. *A specification is said to be* finite *if and only if any flat specification* $\langle \Sigma, \Gamma \rangle$ *occurring as part of a structured specification satisfies that* Γ *is finite.*

Proposition 1. *Let* $\mathbb{I} = \langle \mathsf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \{\vdash^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|}, \{\models^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|} \rangle$ *be the logic of full proper closure fork algebras with urelements. Then,*

1. *if we restrict the morphisms in* $|\mathsf{Sign}|$ *to be injections,* $\mathbb{I}$ *has the weak interpolation property,*
2. *if we restrict* $|\mathsf{Th}_0|$ *to finite presentations, then for any* $t'_1 : \Sigma_1 \to \Sigma', t'_2 : \Sigma_2 \to \Sigma'$ *pushout in* Sign *for* $t_1 : \Sigma \to \Sigma_1, t_2 : \Sigma \to \Sigma_2$, $\varphi_i \in \mathbf{Sen}(\Sigma_i)$ *for* $i = 1, 2$ *such that* $\mathbf{Sen}(t'_1)(\varphi_1) \models^{\Sigma'} \mathbf{Sen}(t'_2)(\varphi_2)$, *there exists* $\Gamma \subseteq \mathbf{Sen}(\Sigma)$ *such that* $\varphi_1 \models^{\Sigma_1} \mathbf{Sen}(t_1)(\Gamma)$, $\mathbf{Sen}(t_2)(\Gamma) \models^{\Sigma_2} \varphi_2$, *and* Γ *is finite,*
3. $\mathbb{I}$ *has the weak amalgamation property, and*
4. $\mathbb{I}$ *has conjunction and implication.*

Proof. 1. By [20, Coro. 4].

2. By [20, Coro. 4] and considering that theory presentations in $|\mathsf{Th}_0|$ are formed by unconditional equations, and by equational completeness.
3. Let $t'_1 : \Sigma_1 \to \Sigma', t'_2 : \Sigma_2 \to \Sigma'$ be a pushout in Sign for $t_1 : \Sigma \to \Sigma_1, t_2 : \Sigma \to \Sigma_2$, and $\mathcal{M}_1$ and $\mathcal{M}_2$ the models $\langle M_1, \{f_i^{\mathcal{M}_1}\}_{i\in\mathcal{I}_1}\rangle \in |\mathbf{Mod}(\Sigma_1)|$ and $\langle M_2, \{f_i^{\mathcal{M}_2}\}_{i\in\mathcal{I}_2}\rangle \in |\mathbf{Mod}(\Sigma_2)|$ respectively such that $\mathbf{Mod}(t_1)(\mathcal{M}_1) = \mathbf{Mod}(t_2)(\mathcal{M}_2)$. Let $\mathcal{M} = \langle M, \{f_i^{\mathcal{M}}\}_{i\in\mathcal{I}}\rangle \in |\mathbf{Mod}(\Sigma)|$ such that $\mathcal{M} = \mathbf{Mod}(t_1)(\mathcal{M}_1) = \mathbf{Mod}(t_2)(\mathcal{M}_2)$, then define $\mathcal{M}' = \langle M, \{{f'}_i^{\mathcal{M}'}\}_{i\in\mathcal{I}'}\rangle$ such that:
   - $\mathcal{I}' = t'_1(\mathcal{I}_1) \cup t'_2(\mathcal{I}_2)$,
   - ${f'}^{\mathcal{M}'}_{t_1\circ t'_1(i)} = f_i^{\mathcal{M}}$, for all$i \in \mathcal{I}$ (notice that $t_1 \circ t'_1 = t_2 \circ t'_2$,
   - ${f'}^{\mathcal{M}'}_{t'_1(i)} = f_i^{\mathcal{M}_1}$, for all $i \in \mathcal{I}_1/t_1(\mathcal{I})$,
   - ${f'}^{\mathcal{M}'}_{t'_2(i)} = f_i^{\mathcal{M}_2}$, for all $i \in \mathcal{I}_2/t_2(\mathcal{I})$ (as t'_1, t'_2 is a pushout in Sign for t_1, t_2, then $t'_1(\mathcal{I}_1/t_1(\mathcal{I})) \cap t'_2(\mathcal{I}_2/t_2(\mathcal{I})) = \emptyset$).

   Then, by construction $\mathbf{Mod}(t'_1)(\mathcal{M}') = \mathcal{M}_1$ and $\mathbf{Mod}(t'_2)(\mathcal{M}') = \mathcal{M}_2$.
4. By [21, p. 26], and considering that fPCFAU are full, thus simple, any boolean combination of equations α is equivalent to an equation $T(\alpha) = 1$.

Theorem 2. *Let* $\mathbb{I} = \langle \mathsf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \{\vdash^{\Sigma}\}_{\Sigma\in|\mathsf{Sign}|}, \{\models^{\Sigma}\}_{\Sigma\in|\mathsf{Sign}|}\rangle$ *be the logic of full proper closure fork algebras with urelements such that* Sign *is restricted to have only injective morphisms,* $\Sigma \in |\mathsf{Sign}|$*, and SP be a finite specification such that* $\mathbf{Sig}[SP] = \Sigma$*, then for all* $\alpha \in |\mathbf{Sen}(\Sigma)|$ $SP \vdash_{\Sigma} \alpha$ *if and only if* $SP \models_{\Sigma} \alpha$*.*

Proof. The proof follows the lines of [10, Thm. 3.9]. Rather than using the satisfaction of the interpolation property, we use the satisfaction of the weak interpolation property (which holds by Prop. 1.1), the fact that $\mathbb{I}$ has de weak amalgamation property (Prop. 1.3), that $\mathbb{I}$ has conjunction and implication (Prop. 1.4) and that whenever specifications are finite, the interpolants are also finite (Prop. 1.2).

In Thm. 2 we proved that Borzyszkowski's calculus is complete for the logic of fork algebras. Yet the result is limited to finite presentations. This fact establishes a limitation for fork algebras. Having a complete calculus required us having an inference rule with infinitary many hypothesis in order to characterize reflexive-transitive closure. Thus, working with finite specifications limits the properties that can be proved to those that are consequences of a finite number of axioms. This does not mean that we will not be able to prove any property involving reflexive-transitive closure. There are many properties involving this operator for which a finite set of axioms is sufficient. In Sec. 4.2 we will overcome this limitation.

4.2 A Complete Calculus for Infinite Structured Specifications in Fork Algebras

In this section we will present a complete calculus for structured (not necessarily finite) specifications in fork algebras. The completeness of the calculus will depend on the fact full proper closure fork algebras with urelements (the semantic models we are using) have a complete calculus [22].

As we mentioned above, the use of fork algebras extended with reflexive-transitive closure requires a proof theory that does not meet the conditions imposed by Borzyszkowski in [10]. The calculus we will present will provide the methodological insight in order to carry out proofs in the infinitary setting.

The following definition presents the calculus for infinite structured specifications. It differs from the calculus presented in Def. 19 in two ways: *a*) we added a rule (*[equiv]*) allowing to replace a specification by another, provided that they are equivalent, and *b*) rules *[CR]*, *[sum1]* and *[sum2]* were replaced by a single, and slightly more complex, rule for $\cup$ (*[sum]*).

Definition 21. *Let* $\mathbb{I} = \langle \mathsf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \{\vdash^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|}, \{\models^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|}\rangle$ *be the logic of full proper closure fork algebras with urelements. Then, the following rules, define a* Sign*-indexed family of entailment relations.*

$$\frac{\Gamma \vdash^{\Sigma} \varphi}{\langle \Sigma, \Gamma\rangle \vdash_{\Sigma} \varphi}\ \textit{[basic]} \qquad \frac{SP_2 \vdash_{\Sigma} \varphi \qquad SP_1 \equiv SP_2}{SP_1 \vdash_{\Sigma} \varphi}\ \textit{[equiv]}$$

$$\frac{SP' \vdash_{\Sigma'} \mathbf{Sen}(\sigma)(\varphi)}{\textbf{derive from } SP' \textbf{ by } \sigma \vdash_{\Sigma} \varphi}\ \textit{[derive]}$$

$$\frac{\{SP_1 \vdash_{\Sigma} \psi\}_{\psi \in \Delta} \qquad \langle \Sigma, \Delta\rangle \cup SP_2 \vdash_{\Sigma} \varphi}{SP_1 \cup SP_2 \vdash_{\Sigma} \varphi}\ \textit{[sum]}$$

$$\frac{SP \vdash_{\Sigma} \varphi}{\textbf{translate } SP \textbf{ by } \sigma \vdash_{\Sigma'} \mathbf{Sen}(\sigma)(\varphi)}\ \textit{[translate]}$$

Theorem 3. *Let* $\langle \mathsf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \{\vdash^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|}, \{\models^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|}\rangle$ *be the logic of full proper closure fork algebras with urelements. Let* $\Sigma, \Sigma' \in |\mathsf{Sign}|$, $\Gamma' \in \mathbf{Sen}(\Sigma')$, $\sigma : \Sigma \to \Sigma'$ *a morphism in* $|\mathsf{Sign}|$, $SP \in \mathsf{Spec}_{\Sigma}$ *and* $\langle \Sigma, \Gamma\rangle \in |\mathsf{Th}_0|$. *Then,* $SP \vdash_{\Sigma} \varphi$ *if and only if* $SP \models_{\Sigma} \varphi$.

Proof. The proof of soundness (i.e. that if $\langle \Sigma, \Gamma\rangle \cup SP \vdash_{\Sigma} \varphi$, then $\langle \Sigma, \Gamma\rangle \cup SP \models_{\Sigma} \varphi$) follows by observing that by Def. 16 all the rules are sound. Completeness trivially follows using the rules *[derive]*, *[equiv]* (considering the fact that $\mathbf{nf}(SP) \equiv SP$), and *[basic]*.

Observing the proof a question arises. Is a calculus like this of any utility? This question has two possible answers. From a theoretical point of view the completeness of this calculus reduces directly to the completeness of the calculus of the underlying logic, thus proofs using the structure building operators reduce to proofs using the calculus for flat specifications. From a practical perspective, theorem proving is an essential tool for formally verifying the behavior of software systems. From this methodological point of view the calculus can be very useful in guiding an engineer in proving properties of structured specifications. We stick to the second argument to explain the changes we introduced in Def. 21 with respect to Def. 19.

Borzyszkowski's completeness proof [10] suggests that the proofs of properties of a union of specifications should be organized by resorting to rules *[CR]*, *[sum1]* and *[sum2]*. This is possible because there is an implicit use of the interpolation property in the elimination of the union of two specifications. In this sense, interpolation is a strong requirement in Borzyszkowski's calculus, even when it is disguised as a combination of weaker properties. In the case of a logic that does not meet this condition, that construction is not possible because the interpolant is not a formula, but a (possibly infinite) set of formulas. To solve this we added rule *[sum]*, which explicits the construction used by Borzyszkowski. On the negative side, the use of the rule *[sum]* eliminates a union between two structured specifications, but introduces another one between a structured specification and a flat one. This responds to the need of using the (possibly infinite) interpolant to complete the proof. This newly added union prevents us from using the structure of the specification as a guide to develop the proof. The inclusion of the rule *[equiv]* gives us a solution (besides the fact that it is needed in the proof of completeness), by enabling the replacement of a given specification by an equivalent one in which the structure can be useful in developing the proof. The next proposition shows some examples of the possibilities that the use of this rule provides.

Proposition 2. [Properties of SBOs]
Let $\mathbb{I} = \langle \mathsf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \{\vdash^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|}, \{\models^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|} \rangle$ *be the logic of full proper closure fork algebras with urelements.*

1. $\langle \Sigma', \mathbf{Sen}(\sigma)(\Gamma) \rangle \cup$ **translate** SP **by** $\sigma \equiv$ **translate** $\langle \Sigma, \Gamma \rangle \cup SP$ **by** σ,
2. **derive from** $\langle \Sigma', \mathbf{Sen}(\sigma)(\Gamma) \rangle \cup SP'$ **by** $\sigma \equiv \langle \Sigma, \Gamma \rangle \cup$**derive from** SP' **by** σ,
3. $\langle \Sigma, \Gamma \rangle \cup (SP_1 \cup SP_2) \equiv (\langle \Sigma, \Gamma \rangle \cup SP_1) \cup SP_2$,
4. $SP_1 \cup SP_2 \equiv SP_2 \cup SP_1$,
5. $(SP_1 \cup SP_2) \cup SP_3 \equiv SP_1 \cup (SP_2 \cup SP_3)$,
6. $\langle \Sigma, \Gamma_1 \rangle \cup \langle \Sigma, \Gamma_2 \rangle \equiv \langle \Sigma, \Gamma_1 \cup \Gamma_2 \rangle$.

Proof. The proofs of 4, 5 and 6 are trivial by Def. 16, and the proof of 3 is an instance of 5. 1 and 2 follows by Def. 16 and set-theoretical reasoning on the classes of models.

We call the reader's attention to the fact that even when we developed this ideas for the particular case of fork algebras, they apply to any language with similar characteristics. Fork algebras served just as a motivation in the formalization of a complete proof calculus for languages that can be considered as "universal" institutions. On the other hand, the reader should notice that the properties presented in Prop. 2 hold, at least, for any institution for which the definition of the operators **Sig**[] and **Mod**[] remain as it was presented in Def. 16.

4.3 A Categorical Characterization of the Structure Building Operations

In this paper we have discussed some problems related to the possibility of having a complete calculus for structured specifications. We now identify another drawback, but this time related to the way structured specifications are defined.

Structured specifications are defined on top of an institution (an appropriate way of formalizing the fact that a specific methodology for software description is defined over an underlying logic). Recalling the definition of structured specifications (as a consequence of the definition of SBOs) and their semantics, it is easy to see that SBOs cannot be characterized by universal constructions in the category Th_0. This is because the operations **derive from** and **translate** alter the relation between the set of axioms in the specification, and its class of models. A problem of this separation is that even when the underlying logic serves as a tool in defining the semantics associated to a structured specification, it is not possible to find a theory (in the underlying logic) which acts as a counterpart of a given structured specification.

The solution we propose is the study of the properties that a logic must have in order to internalize the SBOs as constructions in the category Th_0. In this sense, a possible approach is the search for a logical system in which the effect of applying the operations **derive from** and **translate** on a theory can be characterized by some transformation of the axioms of the specification. Because of the space limitation we will not fully develop this approach in the present work, but we will state some preliminary ideas.

We already mentioned that the motivation behind choosing fork algebras is their expressive power, revealed by the existence of several representation maps from different logics. Fork algebras have a rich model theory in which the logical structure of models is capable of representing the logical structure of models of several logics ubiquitous in software design. For example, [3, Def. 15] shows how first-order dynamic logic is interpreted in fork algebras. In all the extralogical symbols (those that appear in the signature) are interpreted by extending the fork algebraic signature with new constants. As was shown in Def. 8 , fork algebraic constants are interpreted as elements in the domain of models.

Consider the extension of the language of fork algebras presented in Defs. 3 and 4 but replacing the latter definition by the following:

Definition 22. *Let $\mathcal{V}$ be a set of relation variables, then the set Form$(\mathcal{V})$ of first-order ω-CCFAU formulas is the smallest set $\mathcal{F}$ such that:*

- *if $t_1, t_2 \in$ Term$(\mathcal{V})$, then $t_1 = t_2 \in \mathcal{F}$,*
- *if $\alpha, \beta \in \mathcal{F}$ and $C \in \mathcal{V}$, then $\neg\alpha, \alpha \vee \beta, (\exists C)\alpha \in \mathcal{F}$.*

This language is just a first order extension of the original equational one presented in Sec. 2. The main properties presented for the equational version also hold for this new version (for example, the fact that there exists a complete calculus for a concrete class of models in which the domain is formed by binary relations). If we restrict the category of signatures to those that only include relational constants and injective morphisms, mapping $\mathcal{T}$ establishes a relation between structured specifications and flat theories in Th_0. We will use the following definition:

Definition 23. *Let $\Sigma = \langle \{C_i\}_{i \in \mathcal{I}} \rangle$, $\Sigma' = \langle \{C'_i\}_{i \in \mathcal{I}'} \rangle$ and $\sigma : \Sigma \to \Sigma'$. We denote by* $\mathbf{Sen}(\sigma)^\star$ *the backward translation from Σ'-formulas to Σ-formulas* *such that for all $C'_i \in \Sigma'$, if there exists $j \in \mathcal{I}$ such that $\sigma(j) = i$, then C'_i is replaced by C_j in the translation, and left as C'_i otherwise.*

Definition 24. $\mathbf{Ax} : \mathsf{Th}_0 \to \mathsf{Set}$ *is the forgetful functor that for any theory in* Th_0 *returns its corresponding set of axioms.*

Definition 25. *Let* $\mathbb{I} = \langle \mathsf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \{\vdash^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|}, \{\models^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|} \rangle$ *be the logic of full proper closure fork algebras with urelements,* $\Sigma = \langle \{C_i\}_{i \in \mathcal{I}} \rangle$ *and* $\Sigma' = \langle \{C'_i\}_{i \in \mathcal{I}'} \rangle$ *signatures in* $|\mathsf{Sign}|$*, and* $\sigma : \Sigma \to \Sigma'$ *a morphism in* Sign.

- $\mathcal{T}(\langle \Sigma, \Gamma \rangle) = \langle \Sigma, \Gamma \rangle$,
- $\mathcal{T}(\langle \Sigma, \Gamma_1 \rangle \cup \langle \Sigma, \Gamma_2 \rangle) = \langle \Sigma, \Gamma_1 \cup \Gamma_2 \rangle$[6],
- $\mathcal{T}(\textbf{derive from } SP' \textbf{ by } \sigma) = \langle \Sigma, \widehat{\Gamma} \rangle$, *where*

$$\widehat{\Gamma} = Ax \cup \left\{ \left(\exists \{C'_i\}_{i \in \mathcal{I}'/\sigma(\mathcal{I})} \right) \bigwedge_{\alpha \in \mathbf{Sen}(\sigma)^{\star}(\mathbf{Ax}(\mathcal{T}(SP))/Ax)} \alpha \right\} \quad (1)$$

$$Ax = \mathbf{Sen}(\sigma)^{\star}(\mathbf{Ax}(\mathcal{T}(SP)) \cap \mathbf{Sen}(\langle \{C'_{\sigma(i)}\}_{i \in \mathcal{I}} \rangle)) \quad (2)$$

  Equation 2 characterizes those axioms in the structured specification that are expressible over Σ symbols (i.e. those symbols that were not hidden by the operation). Equation 1 states that the set of axioms of the resulting theory is formed by the formulas in Ax and an existential formula replacing those axioms that were not expressible over Σ.
- $\mathcal{T}(\textbf{translate } SP \textbf{ by } \sigma) = \langle \Sigma', \mathbf{Sen}(\sigma)(\mathbf{Ax}(\mathcal{T}(SP))) \rangle$.

Using the previous definition it is easy to prove the following theorem.

Theorem 4. *Let* $\mathbb{I} = \langle \mathsf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \{\vdash^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|}, \{\models^{\Sigma}\}_{\Sigma \in |\mathsf{Sign}|} \rangle$ *be the logic of full proper closure fork algebras with urelements. Let* $\Sigma \in |\mathsf{Sign}|$*, and* SP *a* Σ*-specification. Then,* $\mathbf{Mod}[SP] = \mathbf{Mod}(\mathcal{T}(SP))$.

A more general result can be obtained if we observe that this result is a direct consequence of the fact that the calculus associated with the underlying institution supports an extension satisfying that: *a*) the semantics of the extralogical symbols are elements of a sort of the models (in our case extralogical symbols are constants and their interpretation are binary relations), and *b*) has some kind of existential quantifier and conjunction.

5 Conclusions

Motivated by the use of extensions of fork algebras we analyzed the work of Borzyszkowski on structured specifications and showed that the conditions imposed to logical systems in order to have a complete calculus are too restrictive.

[6] Notice that $\langle \Sigma, \Gamma_1 \cup \Gamma_2 \rangle$ is the apex of the colimit of the diagram formed by the theories $\langle \Sigma, \Gamma_1 \rangle$, $\langle \Sigma, \Gamma_2 \rangle$, $\langle \Sigma, \emptyset \rangle$, and the morphisms $id_{\Sigma 1} : \langle \Sigma, \emptyset \rangle \to \langle \Sigma, \Gamma_1 \rangle$ and $id_{\Sigma 2} : \langle \Sigma, \emptyset \rangle \to \langle \Sigma, \Gamma_2 \rangle$ in Th_0.

We also presented a complete calculus for structured specifications over the logic of fork algebras. Finally, we described some ideas on how to completely characterize structured specifications in the underlying logic. This last topic was treated in a superficial way due to space limitations but we believe that work in this direction will contribute to characterize in an appropriate way the relation between structured specifications and their underlying logic.

References

1. Booch, G., Rumbaugh, J., Jacobson, I.: The unified modeling language user guide. Addison–Wesley Longman Publishing Co., Inc., Boston (1998)
2. Frias, M.F.: Fork algebras in algebra, logic and computer science. Advances in logic, vol. 2. World Scientific Publishing Co., Singapore (2002)
3. Frias, M.F., Baum, G.A., Maibaum, T.S.E.: Interpretability of first-order dynamic logic in a relational calculus. In: de Swart, H. (ed.) RelMiCS 2001. LNCS, vol. 2561, pp. 66–80. Springer, Heidelberg (2002)
4. Frias, M.F., Lopez Pombo, C.G.: Time is on my side. In: Procs. of RelMiCS 7, pp. 105–111 (2003)
5. Frias, M.F., Lopez Pombo, C.G.: Interpretability of first-order linear temporal logics in fork algebras. JLAP 66(2), 161–184 (2006)
6. Goguen, J.A., Burstall, R.M.: Introducing institutions. In: Hutchison, D., Shepherd, W.D., Mariani, J.A. (eds.) Local Area Networks: An Advanced Course. LNCS, vol. 184, pp. 221–256. Springer, Heidelberg (1985)
7. Meseguer, J.: General logics. In: Procs. of the Logic Colloquium 1987, vol. 129, pp. 275–329. North Holland, Amsterdam (1989)
8. Tarlecki, A.: Moving between logical systems. In: Haveraaen, M., Dahl, O.-J., Owe, O. (eds.) Abstract Data Types 1995 and COMPASS 1995. LNCS, vol. 1130, pp. 478–502. Springer, Heidelberg (1996)
9. Lopez Pombo, C.G., Frias, M.F.: Fork algebras as a sufficiently rich universal institution. In: Johnson, M., Vene, V. (eds.) AMAST 2006. LNCS, vol. 4019, pp. 235–247. Springer, Heidelberg (2006)
10. Borzyszkowski, T.: Logical systems for structured specifications. TCS 286, 197–245 (2002)
11. Tarski, A.: On the calculus of relations. JSL 6(3), 73–89 (1941)
12. Maddux, R.D.: Finitary algebraic logic. Zeitschrift fur Mathematisch Logik und Grundlagen der Mathematik 35, 321–332 (1989)
13. Burris, S., Sankappanavar, H.P.: A course in universal algebra. Graduate Texts in Mathematics. Springer, Berlin (1981)
14. Sannella, D., Tarlecki, A.: Specifications in an arbitrary institution. Information and computation 76(2-3), 165–210 (1988)
15. Tarlecki, A.: Abstract specification theory: an overview. In: Procs. of the NATO Advanced Study Institute on Models, Algebras and Logic of Engineering Software. NATO Science Series, pp. 43–79. IOS Press, Marktoberdorf (2003)
16. Diaconescu, R. (ed.): Institution-independent Model Theory, Studies in Universal Logic, vol. 2. Birkhäuser, Basel
17. Mossakowski, T., Maeder, C., Luttich, K.: The heterogeneous tool set, Hets. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 519–522. Springer, Heidelberg (2007)

18. Tarlecki, A.: Towards heterogeneous specifications. In: Gabbay, D., de Rijke, M. (eds.) Frontiers of Combining Systems. Studies in Logic and Computation, vol. 2, pp. 337–360. Research Studies Press (2000)
19. Cengarle, M.V., Knapp, A., Tarlecki, A., Wirsing, M.: A heterogeneous approach to UML semantics. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 383–402. Springer, Heidelberg (2008)
20. Roşu, G., Goguen, J.A.: On equational craig interpolation. Journal of Universal Computer Science 6(1), 194–200 (2000)
21. Tarski, A., Givant, S.: A formalization of set theory without variables. American Mathematical Society Colloqium Publications, Providence (1987)
22. Lopez Pombo, C.G.: Fork algebras as a tool for reasoning across heterogeneous specifications. PhD thesis, Universidad de Buenos Aires (2007)

Towards Managing Dynamic Reconfiguration of Software Systems in a Categorical Setting

Pablo F. Castro[1], Nazareno M. Aguirre[1],
Carlos Gustavo López Pombo[2], and Thomas S.E. Maibaum[3]

[1] Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto and CONICET, Río Cuarto, Córdoba, Argentina
{pcastro,naguirre}@dc.exa.unrc.edu.ar
[2] Departamento de Computación, FCEyN, Universidad de Buenos Aires and CONICET, Buenos Aires, Argentina
clpombo@dc.uba.ar
[3] Department of Computing & Software, McMaster University, Hamilton (ON), Canada
tom@maibaum.org

Abstract. Dynamic reconfiguration, understood as the ability to manage at run time the live components and how these interact in a system, is a feature that is crucial in various languages and computing paradigms, in particular in object orientation. In this paper, we study a categorical approach for characterising dynamic reconfiguration in a logical specification language. The approach is based on the notion of *institution*, which enables us to work in an abstract, logic independent, setting. Furthermore, our formalisation makes use of *representation maps* in order to relate the generic specification of components (e.g., as specified through classes) to the behaviour of actual instances in a dynamic environment. We present the essential characteristics for dealing with dynamic reconfiguration in a logical specification language, indicating their technical and practical motivations. As a motivational example, we use a temporal logic, component based formalism, but the analysis is general enough to be applied to other logics. Moreover, the use of representation maps in the formalisation allows for the combination of different logics for different purposes in the specification. We illustrate the ideas with a simple specification of a Producer-Consumer component based system.

1 Introduction

Modularisation is a key mechanism for dealing with the complexity and size of software systems. It is generally understood as the process of dividing a system specification or implementation into *modules* or *components*, which leads to a structural view of systems, and systems' structure, or *architecture* [10]. Besides its crucial relevance for managing the complexity of systems, the systems' architectural structure also plays an important role in the functional and non functional characteristics of systems. The system architecture has traditionally been *static*, in the sense that it does not change at run time. However, many component based specifications or implementations require dealing with dynamic creation and deletion of components. This is the case, for instance, in *object oriented programming*, where the ability of creating and deleting *objects* dynamically, i.e., at run time, is an intrinsic characteristic. Also in other more abstract contexts, such as software architecture, it is often required to be able to dynamically reconfigure systems, involving in many cases the dynamic creation or deletion of components and connectors [17]. Also in some fields related to fault tolerance, such as self healing and self adaptive systems, it is often necessary to perform dynamic reconfigurations in order to take a system from an inconsistent state back to an acceptable configuration.

Category theory has been regarded as an adequate foundation for formally characterising different notions of components, and component compositions. For instance, in the context of algebraic specification, category theory has enabled the formal characterisation of different kinds of specification extensions [6] . Also, in the context of parallel program design, category theory has been employed for formalising the notion of *superposition*, and the synchronisation of components [7]. In this paper, we present a categorical characterisation of the elements of component composition necessary when dealing with dynamic creation and deletion of components. The characterisation is developed around the notion of an *institution*, which enables us to work in an abstract, logic independent, setting. Furthermore, our formalisation makes use of *representation maps* [22] in order to relate the generic specification of components (e.g., as specified through classes) to the behaviour of actual instances in a dynamic environment. The use of representation maps provides an additional advantage, namely that it allows for the combination of different logics for different purposes in the specification. For instance, one might use a logic for characterising datatypes (e.g., equational logic), another for specifying components (e.g., propositional LTL), and another (e.g., first order LTL) for the description of dynamically reconfigurable systems, involving these components and datatypes.

We present the essential characteristics for dealing with dynamic reconfiguration in a logical specification language, indicating their technical and practical motivations. The approach presented is motivated by the view of system composition as a colimit of a categorical diagram representing the system's structure [3]. Moreover, our approach, as presented in this paper (and in particular due to the logic employed for illustrating the ideas), can be seen as an adaptation of the ideas presented in Fiadeiro and Maibaum's approach to concurrent system specification [7], where the system's structure is inherently rigid, to support dynamic creation/deletion of components, and changes in their interactions. As a motivating example, we use a temporal logic, component based formalism, but the analysis is general enough to be applied to other logics. We will use a single logic for the different parts of a specification, although, as we mentioned, the approach enables one to use different logics for different purposes in specification, as it will be made clearer later on. One might benefit from this fact, in particular for analysis purposes, as it will be argued in the paper. We also discuss some related work. The main ideas presented in the paper are illustrated with a simple specification of a (dynamic) Producer-Consumer component based system.

2 A Motivating Example

In this section, we introduce an example that will be used as a motivation for the work presented in the paper. This example is a simple specification of a Producer-Consumer component based system. The specification is written in linear temporal logic. We assume the reader is familiar with first order logic and linear temporal logic, as well as some basic concepts from category theory [20].

Let us consider a simple Producer-Consumer system, in which two components interact. One of these is a producer, which produces messages (items) that are sent to the other component, the consumer. For simplicity, we assume that the messages communicated are simply bits. The producer's state might then be defined by a bit-typed field *p-current* to hold a produced element, a *boolean* variable *p-waiting* to indicate whether an item is already produced and ready to be sent (so that null values for items are not necessary), and a boolean read variable *ready-in*, so that a producer is informed if the environment is ready to receive a product. We can specify a producer axiomatically, as shown in Figure 1. This specification consists of a set of sorts (Bit and Bool, in this case), a set of fields, some of which are supposed to be controlled by the environment, and a set of action symbols. The axioms of the specification are linear temporal logic formulae characterising the behaviour of the component, in a rather obvious way. Notice Axiom 8, which differentiates local fields from read variables. This is a locality axiom, as in [7], a frame condition indicating that local fields can only be altered by local actions. The axioms of the specification can be thought of as originating in an action language, such as the SMV language, for instance. Notice that the logic used in this specification is *propositional*, which would enable one to algorithmically check properties of producers, by means of model checking tools. A consumer component can be specified in a similar way, as shown in Figure 2.

Component: *Producer*
Read Variables: *ready-in*: Bool
Attributes: *p-current*: Bit, *p-waiting*: Bool
Actions: *produce-0*, *produce-1*, *send-0*, *send-1*, *p-init*
Axioms:
1. $\Box(\textit{p-init} \rightarrow \bigcirc(\textit{p-current} = 0 \wedge \neg \textit{p-waiting}))$
2. $\Box(\textit{produce-0} \vee \textit{produce-1} \rightarrow \neg \textit{p-waiting} \wedge \bigcirc \textit{p-waiting})$
3. $\Box(\textit{produce-0} \rightarrow \bigcirc(\textit{p-current} = 0))$
4. $\Box(\textit{produce-1} \rightarrow \bigcirc(\textit{p-current} = 1))$
5. $\Box((\textit{send-0} \rightarrow \textit{p-current} = \textit{0}) \wedge (\textit{send-1} \rightarrow \textit{p-current} = \textit{1}))$
6. $\Box(\textit{send-0} \vee \textit{send-1} \rightarrow \textit{p-waiting} \wedge \bigcirc \neg \textit{p-waiting})$
7. $\Box(\textit{send-0} \vee \textit{send-1} \rightarrow \textit{p-current} = \bigcirc \textit{p-current})$
8. $\Box(\textit{send-0} \vee \textit{send-1} \vee \textit{produce-0} \vee \textit{produce-1} \vee \textit{p-init} \vee$
$(\textit{p-current} = \bigcirc \textit{p-current} \wedge \textit{p-waiting} = \bigcirc \textit{p-waiting}))$

Fig. 1. A linear temporal logic specification of a simple producer

Component: **Consumer**
Read Variables: *ready-ext*: Bool
Attributes: *c-current*: Bit, *c-waiting*: Bool
Actions: *consume*, *extract-0*, *extract-1*, *c-init*
Axioms:
1. $\Box(\textit{c-init} \rightarrow \bigcirc(\textit{c-current} = 0 \wedge \textit{c-waiting}))$
2. $\Box(\textit{extract-0} \vee \textit{extract-1} \rightarrow \neg \textit{c-waiting} \wedge \textit{c-waiting} \wedge \textit{ready-ext})$
3. $\Box(\textit{extract-0} \rightarrow \bigcirc(\textit{c-current} = 0))$
4. $\Box(\textit{extract-1} \rightarrow \bigcirc(\textit{c-current} = 1))$
5. $\Box(\textit{consume} \rightarrow \neg \textit{c-waiting} \wedge \bigcirc \textit{c-waiting})$
6. $\Box(\textit{consume} \rightarrow \textit{c-current} = \bigcirc \textit{c-current})$
7. $\Box(\textit{consume} \vee \textit{extract-0} \vee \textit{extract-1} \vee \textit{c-init} \vee$
$(\textit{c-current} = \bigcirc \textit{c-current} \wedge \textit{c-waiting} = \bigcirc \textit{c-waiting}))$

Fig. 2. A linear temporal logic specification of a simple consumer

A mechanism for putting these specifications together is by coordinating them, for instance, by indicating how read variables are "connected" or identified with fields of other components, and by synchronising actions. Basic action synchronisation can be employed for defining more sophisticated forms of interaction, e.g., procedure calls. In [7,8], the described form of coordination between components is achieved by the use of "channels"; a channel is a specification with no axioms, but only symbol declarations, together with two mappings, identifying the symbols in the channel specification with the actions to be synchronised and the fields/variables to be identified, in the corresponding components. For our example, we would want to make the components interact by synchronising the `send-i` and `extract-i` actions, of the producer and consumer, respectively, and by identifying `ready-in` and `p-waiting`, in the producer, with `c-waiting` and `ready-ext` in the consumer, respectively.

It is known that specifications and symbol mappings in this logic form a category which admits finite colimits [12]. This is important due to the fact that the above described coordination between producers and consumers, materialised as a "channel", forms the following *diagram* (in the categorical sense):

$$\begin{tikzcd}
\textbf{Producer} & & \textbf{Consumer} \\
& {[\textbf{vars} : \mathbf{x}, \mathbf{y} : \textbf{Bool}; \textbf{acts} : \mathbf{a}, \mathbf{b}]} \arrow[ul, "\begin{array}{c} x \mapsto \textit{ready-in} \\ y \mapsto \textit{p-waiting} \\ a \mapsto \textit{send-0} \\ b \mapsto \textit{send-1} \end{array}"'] \arrow[ur, "\begin{array}{c} x \to \textit{c-waiting} \\ y \to \textit{ready-ext} \\ a \to \textit{extract-0} \\ b \to \textit{extract-1} \end{array}"] &
\end{tikzcd}$$

The colimit object for this diagram is a specification that corresponds to the combined behaviour of the producer and consumer, interacting as the diagram indicates.

The architecture of the system, represented by the diagram, clearly does not directly admit reconfiguration. More precisely, how components are put together is prescribed in an external way with respect to component definition (by the construction of a diagram), and although the represented specification can be constructed as a colimit, the possibility of having a component managing the population of instances of other components (as an example of dynamic reconfiguration, motivated by what is common in object orientation) is not compatible with the way configurations are handled in this categorical approach.

Our aim in this paper is to provide a categorical characterisation of a generalisation of the above situation, when both the population of live components and their connections are manipulated, within a system, dynamically. The actual way in which these elements (components and connections) are dynamically manipulated depends on the particular problem or system being specified. For instance, we might have a system where the number of components is maintained over time, but the way in which these components interact is changed dynamically. Alternatively, we might have a system in which a certain kind of component, e.g., clients, are created dynamically, but there is a fixed number of servers.

3 Dynamic Reconfiguration in an Institutional Setting

In this section, we present our proposal for formally characterising dynamic reconfiguration in a logical specification language. In order to make the approach generic (i.e., to make it applicable to a wide range of logics and related formalisms), we develop the formalisation using the notion of *institution*. This enables us to present the formalisation in a high level, logic independent, setting. The theory of institutions was presented by Goguen and Burstall in [11]. Institutions provide a formal and generic definition of what a logical system is, and of how specifications in a logical system can be structured [21]. Institutions have evolved in a number of directions, from an abstract theory of software specification and development [25] to a very general version of abstract model theory [5], and offered a suitable formal framework for addressing heterogeneity [19,24], including applications related to widely used (informal) languages, such as the UML [4]. The following definitions were taken from [18].

Definition 1. [Institution]
An institution *is a structure of the form* $\langle \mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \{\models^{\Sigma}\}_{\Sigma \in |\mathbf{Sign}|}\rangle$ *satisfying the following conditions:*

- $\mathbf{Sign}$ *is a category of signatures,*
- $\mathbf{Sen} : \mathbf{Sign} \to \mathbf{Set}$ *is a functor (let* $\Sigma \in |\mathbf{Sign}|$[1]*, then* $\mathbf{Sen}(\Sigma)$ *returns the set of* Σ*-sentences),*
- $\mathbf{Mod} : \mathbf{Sign}^{\mathsf{op}} \to \mathbf{Cat}$ *is a functor (let* $\Sigma \in |\mathbf{Sign}|$*, then* $\mathbf{Mod}(\Sigma)$ *returns the category of* Σ*-models),*
- $\{\models^{\Sigma}\}_{\Sigma \in |\mathbf{Sign}|}$*, where* $\models^{\Sigma} \subseteq |\mathbf{Mod}(\Sigma)| \times \mathbf{Sen}(\Sigma)$*, is a family of binary relations,*

[1] $|\mathbf{Sign}|$ denotes the class of objects of category $\mathbf{Sign}$.

and for any signature morphism $\sigma : \Sigma \to \Sigma'$, Σ-sentence $\phi \in \mathbf{Sen}(\Sigma)$ and Σ'-model $\mathcal{M}' \in |\mathbf{Mod}(\Sigma)|$ the following $\models$-invariance condition holds:

$$\mathcal{M}' \models^{\Sigma'} \mathbf{Sen}(\sigma)(\phi) \quad \textit{iff} \quad \mathbf{Mod}(\sigma^{\mathsf{op}})(\mathcal{M}') \models^{\Sigma} \phi \; .$$

Institutions are an abstract formulation of the notion of logical system where the concepts of languages, models and truth are characterised using category theory. Roughly speaking, an institution is made up of a category **Sign** which defines the syntax of the logic in terms of the possible vocabularies and translations between them, a functor $\mathbf{Sen} : \mathbf{Sign} \to \mathbf{Set}$ that captures the way in which formulae are built from vocabularies (this functor maps translations between vocabularies to translations between sets of formulae in the obvious way). Moreover, the semantical part of a given logic is captured using a covariant functor $\mathbf{Mod} : \mathbf{Sign}^{\mathsf{op}} \to \mathbf{Cat}$ which maps each vocabulary to the category of its possible models. This functor is covariant since any translation of symbols uniquely determines a model reduct. Finally, an indexed relation $\models^{\Sigma}$: $\mathbf{Mod}(\Sigma) \times \mathbf{Sen}(\Sigma)$ is used to capture the notion of truth. A restriction is imposed on this relationship to ensure that truth is not affected by change of notation. Examples of institutions are: propositional logic, equational logic, first-order logic, first-order logic with equality, dynamic logics and temporal logics (a detailed list is given in [12]). Note that any of these logics has the four components of institutions. Furthermore, in these logics the notion of truth does not depend on the particular choice of the symbols in a formula, i.e., the truth of a formula depends only on its structure and not on the contingent names of its parts.

The logic we used for specifying components, linear temporal logic, constitutes an institution. Its category of signatures is composed of alphabets (sets of propositional variables, since bit-typed fields are straightforwardly encoded as boolean variables, and action occurrence can directly be represented as boolean variables) as objects, and mappings between alphabets as morphisms. The grammar functor $\mathbf{Sen} : \mathbf{Sign} \to \mathbf{Set}$ for this logic is simply the recursive definition of formulae for a given vocabulary. The functor $\mathbf{Mod} : \mathbf{Sign}^{\mathsf{op}} \to \mathbf{Cat}$ maps signatures (alphabets) to their corresponding classes of models, and alphabet contractions (i.e., reversed alphabet translations) to "reducts". The relationship $\models^{\Sigma}$ is the usual satisfaction relation in LTL. By means of a simple inductive argument, it is rather straightforward to prove that this relationship satisfies the invariance condition, and thus LTL is an institution.

The logic we used so far is propositional. A first-order version of this logic is presented in [16], where variables, function symbols and predicate symbols are incorporated, as usual. This first order linear temporal logic admits a single type of "flexible" (i.e., whose interpretation is state dependent) symbol, namely flexible variables. All other symbols (function and predicate symbols, in particular) are rigid, in the sense that their interpretations are state independent. We will consider a generalisation of this logic, in which function and relation symbols are "split" into flexible and rigid (notice that flexible variables become a special case of flexible function symbols). This will simplify our specifications, and the presentation of the ideas in this paper. The propositional specifications given

Component: *ProducerManager*
Read Variables: *ready-in* : NAME $\to$ Bool
Attributes: *p-current* : NAME $\to$ Bit,
p-waiting : NAME $\to$ Bool
Actions: *produce-0*(n: NAME), *produce-1*(n: NAME), *send-0*(n: NAME),
send-1(n: NAME), *p-init*(n: NAME)
Axioms:
1. $\Box(\forall n \in \text{NAME} : \textit{p-init}(n) \to \bigcirc(\textit{p-current}(n) = 0 \land \neg \textit{p-waiting}(n)))$
2. $\Box((\forall n \in \text{NAME} : \textit{produce-0}(n) \lor \textit{produce-1}(n) \to \neg \textit{p-waiting}(n) \land \bigcirc \textit{p-waiting}(n))$
3. $\Box(\forall n \in \text{NAME} : \textit{produce-0}(n) \to \bigcirc(\textit{p-current}(n) = 0))$
4. $\Box(\forall n \in \text{NAME} : \textit{produce-1}(n) \to \bigcirc(\textit{p-current}(n) = 1))$
5. $\Box(\forall n \in \text{NAME} : (\textit{send-0}(n) \to \textit{p-current(n) = 0}) \land (\textit{send-1}(n) \to \textit{p-current(n) = 1}))$
6. $\Box(\forall n \in \text{NAME} : \textit{send-0}(n) \lor \textit{send-1}(n) \to \textit{p-waiting}(n) \land \bigcirc \neg \textit{p-waiting}(n))$
7. $\Box(\forall n \in \text{NAME} : \textit{send-0}(n) \lor \textit{send-1}(n) \to \textit{p-current}(n) = \bigcirc \textit{p-current}(n))$
8. $\Box(\forall n \in \text{NAME} : \textit{send-0}(n) \lor \textit{send-1}(n) \lor \textit{produce-0}(n) \lor \textit{produce-1}(n) \lor \textit{p-init}(n) \lor (\textit{p-current}(n) = \bigcirc \textit{p-current}(n) \land \textit{p-waiting}(n) = \bigcirc \textit{p-waiting}(n)))$

Fig. 3. A first-order linear temporal logic specification of a producer manager

before can be thought of as first-order specifications, where the "first-order" elements of the language are not used. By employing the ideas presented in [21], we can prove in a straightforward way that this first-order linear temporal logic is also an institution.

A specification is essentially a theory presentation, as usually defined [12,18]. Any category of alphabets and translations can be lifted to categories **Th** and **Pres**, of theories and theory presentations, where morphisms are theorem and axiom preserving translations, respectively [9]. The relationships between these categories and **Sign** are materialised as forgetful functors (which reflect colimits).

A traditional way of dealing with dynamic reconfiguration is by specifying *managers* of components. A manager of a component C is a specification which intuitively provides the behaviour of various instances of C, and usually enables the manipulation of instances of C. For example, for our Producer specification, a manager might look as in Figure 3. Notice that we are using the "first-order" expressive power of the language in this specification.

Notice the clear relationship between our producer specification and the specification of a manager of producers. With respect to the syntax (i.e., the symbols used in the specification), the manager is a *relativisation* of the producer, in which all variables and actions incorporate a new parameter, namely, the "instance" to which the variable belongs, or the action to which it is applied, correspondingly. A first intuition would be to try to characterise the relationship between the signature of a component and the signature of its manager as a signature morphism. However, such a relationship is not possible, since signature morphisms must preserve the arities of symbols, and arities are not preserved in managers. A similar situation is observed with formulae in the theory presentation. It is clear that all axioms of Producer are somehow "preserved" in the producer manager, since what we want to capture is the fact that all (live) producer instances behave as the producer specification indicates. A way to solve this problem, the mismatch between the notion of signature morphism and what is needed for capturing the component-manager relationship, would be to redefine the notion of signature morphism, so that new parameters are allowed when a symbol is translated. We have attempted this approach, which led to a complicated, badly structured, characterisation [2]. In particular, redefining the notion of signature morphism forced us to redo many parts of the traditional definition of institutions. In this paper, we present a different characterisation, which is much simpler and better structured. In this approach, we do not characterise the relationship between components as managers *within* an institution, but *outside* institutions, employing the notion of *representation map* [22].

As we explained before, the static description of components is given in **Pres**, the category of theory presentations (in first-order linear temporal logic), where the objects of the category define the syntax (signature) and axioms characterising component behaviour. Diagrams in this category correspond to component based designs, indicating the way components interact in a system, and colimits of these diagrams correspond to the behaviour of the structured design, "linked" as a monolithic component (the colimit object). These diagrams, and their colimits, characterise *static* composition, in a suitable way. In order to provide a dynamic behaviour associated with components, we start by constructing *managers* of components, as we illustrated for producers. First, let us consider an endofunctor $(-)^M : \mathbf{Sign} \rightarrow \mathbf{Sign}$, which maps each signature Σ to the signature Σ^M, obtained simply by incorporating a new sort $\bullet_\Sigma$, and a new parameter of this sort in each of the (flexible) symbols of Σ. Notice that the logic needs to support arguments in symbols (i.e., it needs to provide a notion of parameterisation), since otherwise adding a new parameter to a symbol would not be possible. For the case of first-order linear temporal logic, the functor $(-)^M$ maps a signature $\Sigma = \langle S, V, F_r, F_f, R_r, R_f \rangle$ (where S is the set of sorts, V the set of variables, and F and R the sets of function and predicate symbols, separated into flexible ("f" subscript) and rigid ("r" subscript) symbols) to the signature $\Sigma^M = \langle S^M, V, F_r, F_f^M, R_r, R_f^M \rangle$, where: *(i)* $S^M = S \cup \{\bullet_\Sigma\}$, where $\bullet_\Sigma$ is a sort name such that $\bullet_\Sigma \notin S$, *(ii)* $F_f^M = \{f : \bullet_\Sigma, w \rightarrow s \mid f : w \rightarrow s \in F_f\}$, and *(iii)* $R_f^M = \{r : \bullet_\Sigma, w \mid r : w \in R_f\}$.

Notice that we incorporate the extra parameter only into the symbols that constitute the state of the component. For statically interpreted symbols, the extra parameter is unnecessary. The way in which $(-)^M$ chooses the new sort name $\bullet_\Sigma$ for each Σ is not important for our current purposes. In our example, we chose a new sort NAME, for the identifiers of instances of components. With respect to morphisms, $(-)^M$ maps each signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ to morphism $\sigma^M : \Sigma^M \rightarrow \Sigma'^M$, defined exactly as σ but mapping $\bullet_\Sigma \mapsto \bullet_{\Sigma'}$. Functor $(-)^M$ captures the relationship between components and their managers, via a construction which is external to the category of signatures (i.e., via a functor, not a morphism). In order to capture the relationship that exists between the specification of a component and the specification of the manager of this component, we now define a natural transformation $\eta^M : \mathbf{Sen} \xrightarrow{\cdot} \mathbf{Sen} \circ (-)^M$. This natural transformation corresponds to a mapping $\eta^M_\Sigma : \mathbf{Sen}(\Sigma) \to \mathbf{Sen}(\Sigma^M)$, which maps any formula φ of $\mathbf{Sign}(\Sigma)$ to a formula φ^M of $\mathbf{Sign}(\Sigma^M)$. The definition of φ^M is simple: for each element of the signature which represents part of the "state" appearing in φ (in our case, a flexible symbol), add an extra parameter of type $\bullet_\Sigma$, and universally quantify it. Notice that this requires the logic to be first-order, so that universal quantification is possible. In the case of our specifications, given any formula φ we choose a $\bullet_\Sigma$-labelled variable $x_{\bullet_\Sigma}$. Each occurrence of any flexible function symbol $f : w \to s$ of the form $f(t_1, \ldots, t_n)$ (where $t_1, \ldots, t_n$ are terms) is replaced by the term $f(x_{\bullet_\Sigma}, t_1, \ldots, t_n)$ (note that $f : \bullet_\Sigma, w \to s$ is a flexible function symbol of Σ^M), and similarly for flexible relations. After this step, we obtain a formula φ'. Finally, we define $\varphi^M = (\forall x_{\bullet_\Sigma} \in \bullet_\Sigma : \varphi')$. Again, we have captured the relationship between a component specification and the specification of its corresponding manager *externally*, via a natural transformation, instead of internally, within the category of specifications.

Finally, let us deal with *models*. We define a natural transformation $\gamma : \mathbf{Mod} \circ ((-)^M)^{\mathsf{op}} \xrightarrow{\cdot} \mathbf{Mod}$. That is, we have a natural family of functors $\gamma_\Sigma : \mathbf{Mod}(\Sigma^M) \to \mathbf{Mod}(\Sigma)$, which maps each model M' of $\mathbf{Mod}(\Sigma^M)$ to a model M of $\mathbf{Mod}(\Sigma)$, the model obtained by taking away the parameters of type $\bullet_\Sigma$ in every function and relation in M'. In a similar way, any morphism $m : M' \to M$ of $\mathbf{Mod}(\Sigma^M)$ can be translated to a morphism in $\mathbf{Mod}(\Sigma)$ $\gamma_\Sigma : \gamma_\Sigma(M') \to \gamma_\Sigma(M)$, corresponding to the restriction of m to the set of sorts different from $\bullet_\Sigma$. For the sake of brevity, we skip the detailed definition of these natural transformations.

For our first-order linear temporal logic, we have the following property:

Property 1. For every signature Σ, $\varphi \in \mathbf{Sen}(\Sigma)$ and $M' \in \mathbf{Mod}(\Sigma^M)$ the following holds: $M' \models_{\Sigma^M} \eta_\Sigma(\varphi) \Leftrightarrow \gamma_\Sigma(M') \models_\Sigma \varphi$.

Intuitively, this property states that γ and η preserve the satisfaction relation. This kind of relation between institutions is called a *representation map* [22]. Since the logic for components and for managers is the same, we have an endo-representation map, which relates components with their managers. Let us recall the definition of representation map, as given in [22].

Definition 2. *(Representation map between institutions)*
Let $\langle \mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \{\models_\Sigma\}_{\Sigma \in |\mathbf{Sign}|} \rangle$ *and* $\langle \mathbf{Sign}', \mathbf{Sen}', \mathbf{Mod}', \{\models'_\Sigma\}_{\Sigma \in |\mathbf{Sign}'|} \rangle$ *be the institutions* I *and* I' *respectively, then* $\langle \gamma^{Sign}, \gamma^{Sen}, \gamma^{Mod} \rangle : I \to I'$ *is a* representation map *of institutions if and only if:*

– $\gamma^{Sign} : \mathbf{Sign} \to \mathbf{Sign}'$ *is a functor,*
– $\gamma^{Sen} : \mathbf{Sen} \xrightarrow{\cdot} \mathbf{Sen}' \circ \gamma^{Sign}$*, is a natural transformation (i.e. a natural family of functions* $\gamma^{Sen}_\Sigma : \mathbf{Sen}(\Sigma) \to \mathbf{Sen}'(\gamma^{Sign}(\Sigma))$*), such that for each* $\Sigma_1, \Sigma_2 \in |\mathbf{Sign}|$ *and* $\sigma : \Sigma_1 \to \Sigma_2$ *morphism is* $\mathbf{Sign}$*,*

$$\begin{tikzcd}
\mathbf{Sen}(\Sigma_2) \arrow[r, "\gamma^{Sen}_{\Sigma_2}"] & \mathbf{Sen}'(\gamma^{Sign}(\Sigma_2)) & \Sigma_2 \\
\mathbf{Sen}(\Sigma_1) \arrow[u, "\mathbf{Sen}(\sigma)"'] \arrow[r, "\gamma^{Sen}_{\Sigma_1}"] & \mathbf{Sen}'(\gamma^{Sign}(\Sigma_1)) \arrow[u, "\mathbf{Sen}'(\gamma^{Sign}(\sigma))"'] & \Sigma_1 \arrow[u, "\sigma"']
\end{tikzcd}$$

– $\gamma^{Mod} : \mathbf{Mod}' \circ (\gamma^{Sign})^{\mathsf{op}} \stackrel{\cdot}{\to} \mathbf{Mod}$, *is a natural transformation (i.e. the family of functors* $\gamma^{Mod}_{\Sigma} : \mathbf{Mod}'((\gamma^{Sign})^{\mathsf{op}}(\Sigma)) \to \mathbf{Mod}(\Sigma)$ *is natural), such that for each* $\Sigma_1, \Sigma_2 \in |\mathbf{Sign}|$ *and* $\sigma : \Sigma_1 \to \Sigma_2$ *morphism in* $\mathbf{Sign}$*,*

$$\begin{tikzcd}
\mathbf{Mod}'((\gamma^{Sign})^{\mathsf{op}}(\Sigma_2)) \arrow[r, "\gamma^{Mod}_{\Sigma_2}"] \arrow[d, "\mathbf{Mod}'((\gamma^{Sign})^{\mathsf{op}}(\sigma^{\mathsf{op}}))"] & \mathbf{Mod}(\Sigma_2) \arrow[d, "\mathbf{Mod}(\sigma^{\mathsf{op}})"] & \Sigma_2 \\
\mathbf{Mod}'((\gamma^{Sign})^{\mathsf{op}}(\Sigma_1)) \arrow[r, "\gamma^{Mod}_{\Sigma_1}"] & \mathbf{Mod}(\Sigma_1) & \Sigma_1 \arrow[u, "\sigma"']
\end{tikzcd}$$

such that for any $\Sigma \in |\mathbf{Sign}|$*, the function* $\gamma^{Sen}_{\Sigma} : \mathbf{Sen}(\Sigma) \to \mathbf{Sen}'(\gamma^{Sign}(\Sigma))$ *and the functor* $\gamma^{Mod}_{\Sigma} : \mathbf{Mod}'(\gamma^{Sign}(\Sigma)) \to \mathbf{Mod}(\Sigma)$ *preserves the following satisfaction condition: for any* $\alpha \in \mathbf{Sen}(\Sigma)$ *and* $\mathcal{M}' \in |\mathbf{Mod}(\gamma^{Sign}(\Sigma))|$*,*

$$\mathcal{M}' \models_{\gamma^{Sign}(\Sigma)} \gamma^{Sen}_{\Sigma}(\alpha) \quad \textit{iff} \quad \gamma^{Mod}_{\Sigma}(\mathcal{M}') \models_{\Sigma} \alpha \ .$$

Representation maps have been studied in detail in [22,13]. The intuition that leads us to think that "all instances of a certain component type behave as the component (type) specification indicates" is justified by the following property of representation maps (see [22]):

Property 2. Semantic deduction is preserved by representation maps: for any institution representation $\rho : \mathbf{I} \to \mathbf{I}'$, signature $\Sigma \in |\mathbf{Sign}|$, set $\Phi \subseteq \mathbf{Sen}(\Sigma)$ of Σ-sentences, and Σ-sentence $\varphi \in \mathbf{Sen}(\Sigma)$, if $\Phi \vDash_{\Sigma} \varphi$, then $\rho^{Sen}_{\Sigma}(\Phi) \vDash'_{\rho^{Sign}(\Sigma)} \rho^{Sen}_{\Sigma}(\varphi)$.

Intuitively, this property says that managers of components preserve the properties that the specification of the corresponding components imply. Notice that this is the usual intuition: when one is reasoning about a *class* in object oriented design, one does so thinking of a *generic* template of instances of the class, so that the programmed behaviour will be that of all instances of the class[2]. This construction is also associated with some specification related mechanisms; for instance, the notion of *schema promotion* in Z [27] is captured by this very same notion of representation map. The promoted schemas are obtained via natural transformations from the original set of schemas.

[2] Obviously, classes also define additional behaviour, associated with the manipulation of instances (constructors, destructors, etc.).

4 Managing Dynamic Population and Interaction

In our initial example, a basic structure of a system is given in terms of component specifications, as well as specifications of the interactions between components. We also mentioned that these interactions are materialised as channels, which enable one to define (categorical) diagrams, corresponding to static architectural designs of systems. We have already dealt with part of the generalisation of this situation to allow for dynamic creation/deletion of components, via component managers. We still need to describe the way in which the component population is actually managed, and how instances of components interact. For example, we would need ways of dynamically managing the population of components, and dynamically allocating live producers to live consumers, in the context of our example. In order to achieve the first of these goals, one needs to provide extensions of the manager components, introducing some specific behaviours into the managers (for example, some actions and properties related to the creation or deletion of components at run time). This, of course, needs to be manually specified, whereas the relativisation of component behaviour to instance behaviour is directly handled by the above presented representation map. For the case of our producer manager, such an extension could be the one presented in Figure 4. Notice how a set of live instances is introduced (via a flexible predicate), and how actions for population management can be specified. In this example, we have *new*, which allows us to create new producers. Axiom 10, for instance, indicates that in order to make an instance live, it must be originally "dead", and that *p-init* is executed at creation, on the newly created instance.

Now, let us deal with the connections. In order to dynamically allocate producers to consumers, we define a kind of connection template (we then exploit the previously introduced representation map to build a connection manager). We start by identifying the parts of the components that possibly need to be coordinated. In our case, we identify the fields of producers and consumers that need to be "exported" to other components, and the actions that need to be synchronised. These communicating elements are combined via a coproduct, yielding a vocabulary with parts from producers and parts from consumers, that will be used in order to describe the possible interactions between these types of components. This situation can be generically illustrated as follows:

$$\begin{tikzcd}
\mathbf{C_1} & & \mathbf{Con_1^1} + \mathbf{Con_2^1} & & \mathbf{C_2} \\
& \mathbf{Con_1} \arrow[ul, "m"] \arrow[ur, "i"'] & & \mathbf{Con_2} \arrow[ul, "j"] \arrow[ur, "n"'] & \\
& & 0 \arrow[uu, "!"] \arrow[uull, bend left, "!"] \arrow[uurr, bend right, "!"'] & &
\end{tikzcd}$$

This diagram involves two components, $\mathbf{C_1}$ and $\mathbf{C_2}$. The first component has a communicating language $\mathbf{Con_1}$ and the second component has a communication language $\mathbf{Con_2}$. As stated above, we want to connect these types of components

Component: *ExtendedProducerManager*
Read Variables: *ready-in* : NAME → Bool
Attributes: *producers* : NAME → Bool, *p-current* : NAME → Bit,
p-waiting : NAME → Bool
Actions: *produce-0*(n: NAME), *produce-1*(n: NAME), *send-0*(n: NAME),
send-1(n: NAME), *p-init*(n: NAME), *new*(n: NAME)
Axioms:
... /* axioms of ProducerManager */
9. $\forall n \in \text{NAME} : \neg \mathit{producers}(n)$
10. $\Box(\forall n \in \text{NAME} : \mathit{new}(n) \rightarrow \neg \mathit{producers}(n) \wedge \bigcirc(\mathit{producers}(n) \wedge \mathit{p\text{-}init}(n)))$
11. $\Box(\forall n \in \text{NAME} : \mathit{produce\text{-}0}(n) \rightarrow \mathit{producers}(n))$
...

Fig. 4. An extension of the producer manager, which handles instance creation

using these communication languages. In contrast to our initial (static) example, we do not identify common parts in the components, but use the coproduct of the communicating languages to obtain a language in which to describe the interactions. This also provides more flexibility in the communication definition. In addition, the diagram involves the initial object of category **Sign**, together with the (unique) morphisms ! from this object to the other components. This is necessary since, after applying $(-)^M$, we obtain a component 0^M which has only one sort, and the arrows $!^M$ (obtained applying the functor $(-)^M$ to the arrows !) identify all the new sorts added in the other components by $(-)^M$. The explicit inclusion of the initial object has as a consequence that only one new sort (of component names) is included in the final design.

A suitable connection template for our example could be the one in Figure 5. A manager of this specification is built in the same way that managers of components are constructed. In the same way that we extended the managers of components, we will need to extend the manager of connections, to indicate how connections work. A sample extension of the manager of connections is also shown in Fig. 5. The generic situation depicted in the previous diagram can be expanded, by means of the introduced representation map, to the following:

$$\begin{tikzcd}
\mathbf{MC_1} & & \mathbf{MI} & & \mathbf{MC_2} \\
\mathbf{C_1}^M \arrow[u] & & (\mathbf{Con_1^1} + \mathbf{Con_2^1})^M \arrow[u] & & \mathbf{C_2}^M \arrow[u] \\
& \mathbf{Con_1^M} \arrow[ul, "m^M"] \arrow[ur, "i^M"'] & & \mathbf{Con_2}^M \arrow[ul, "j^M"] \arrow[ur, "n^M"'] & \\
& & 0^M \arrow[uu, "!^M"] \arrow[uull, bend left, "!^M"] \arrow[uurr, bend right, "!^M"'] & &
\end{tikzcd}$$

Component: *ConnectionTemplate*
Attributes: *ready-in* : Bool, *ready-ext* : Bool, *p-waiting* : Bool, *c-waiting* : Bool
Actions: *send-0*, *send-1*, *extract-0*, *extract-1*

Component: *ExtendedConnectionManager*
Attributes: *ready-in* : NAME → Bool, *ready-ext* : NAME → Bool,
p-waiting : NAME → Bool, *c-waiting* : NAME → Bool,
connected : NAME, NAME → Bool
Actions: *send-0*(n: NAME), *send-1*(n: NAME),
extract-0(n: NAME), *extract-1*(n: NAME), *connect*(n, m: NAME)
Axioms:
1. $\Box(\forall n, m \in \text{NAME} : \mathit{connect}(n,m) \rightarrow \neg \mathit{connected}(n,m) \wedge \bigcirc(\mathit{connected}(n,m))$
2. $\Box(\forall n, m \in \text{NAME} : \mathit{connected}(n,m) \rightarrow (\mathit{send\text{-}0}(n) \leftrightarrow \mathit{extract\text{-}0}(m)))$

...

Fig. 5. A connection template, indicating the vocabulary relevant for communication, and an extension of its manager

The specifications C_1^M, $(\mathbf{Con_1^1} + \mathbf{Con_2^1})^M$ and $\mathbf{C_2}^M$ are obtained via the functor $(-)^M$. The components $\mathbf{MC_1}$, $\mathbf{MI}$ and $\mathbf{MC_2}$ are the (ad-hoc) extensions of the manager components. As for the case of static configurations, the colimit of this diagram gives us the final design. It is interesting to note that, since we have used abstract concepts such as institutions and representation maps, the concepts introduced in this section can be instantiated with other logics. In particular, there is no need to use the same logic for component specification and manager specification. Notice that to the extent that these logics can be connected by a representation map, all of the presented characterisation is applicable. For example, we can use a propositional temporal logic to describe the components, taking advantage of decision procedures for such a logic, and use a first-order temporal logic to describe the managers. The representation map between these two logics still enables us to "promote" the properties verified for components (algorithmically, if the logic for component specification is decidable) to properties of all instances of components. Furthermore, we could take this idea even further, and use yet another logic for datatype specification (e.g., a suitable equational logic), and promote the properties of datatypes to components and managers, again by exploiting representation maps.

5 Conclusions

Many specification languages need to deal with dynamic reconfiguration and dynamic population management. In Z, for instance, this is done via schema promotion [27], which is understood simply as a syntactical transformation. In the context of B [1], there is a similar need, in particular when using B as a target language for analysis of object oriented models (e.g., the UML-B approach).

Object oriented extensions of model oriented languages, such as VDM++, Object Z or Z++, have built in mechanisms for dealing with dynamic reconfiguration, as is inherent in object orientation. Other logical languages, for instance some logical languages used for software architectures (e.g., ACME), also require dynamism in specifications. Generally, the mechanisms for dynamism in the mentioned languages are syntactical.

Besides the work mentioned above, there exist some other related approaches, closer to what is presented in this paper. A useful mechanism for formally characterising dynamic reconfiguration is that based on graph transformation, as in [14], which has been successfully applied in the context of dynamic software architectures [26]. As opposed to our work, in this approach the notion of manager is not present, and thus it is less applicable to contexts where this notion is intrinsic (e.g., object orientation, schema promotion, etc.). Another related approach is that of Knapp et al. [15], who present an approach for specifying service-oriented systems, with categorical elements. Knapp et al. employ mappings (from local theories to a global one) for specifying component synchronisation, but composition is not characterised via universal constructions, as in our approach. Another feature of our approach, not present in Knapp et al.'s, is the preservation of the original design's modularisation, via representation maps (notice that each component is mapped to a similar component in a more expressive setting, where its dynamic behaviour is expressed).

We presented the requirements for dealing with dynamic reconfiguration in a logical specification language, in a categorical way. Our categorical characterisation is general enough so that it applies to a wide variety of formalisms, which we have illustrated using a temporal logic. An essential characteristic of the logical system is that quantification is required, so that collections of instances of components can be handled. Our work might help in understanding the relationship between basic "building block" specifications and the combined, whole system, in the presence of dynamic reconfiguration.

We also believe that this work has interesting practical applications. The categorical setting we are working with admits working with different (but related) logics for component specification, and the description of dynamic systems. This might enable one, for instance, to use a less expressive (perhaps decidable) logic for the specification of components, whose specifications could be mapped to more expressive (generally undecidable) logics, where the dynamism of systems is characterised. Understanding the relationships between the different parts of the specification can be exploited for practical reasons, for example for promoting properties verified in components (using for example a decision procedure) to the specification of the system. Also, some recently emerged fields, such as service oriented architectures, require dealing with highly dynamic environments, where formalisations as the one proposed in this paper might be useful. As expressed in [23], there is a clear need for work in the direction of our proposal, so we plan to investigate the applicability of our approach in some of the contexts mentioned in [23].

References

1. Abrial, J.R.: The B Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Aguirre, N., Maibaum, T.: Some Institutional Requirements for Temporal Reasoning about Dynamic Reconfiguration. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 407–435. Springer, Heidelberg (2004)
3. Burstall, R., Goguen, J.: Putting Theories together to make Specifications. In: Proc. of the Fifth International Joint Conference on Artificial Intelligence (1977)
4. Cengarle, M., Knapp, A., Tarlecki, A., Wirsing, M.: A Heterogeneous Approach To UML Semantics. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 383–402. Springer, Heidelberg (2008)
5. Diaconescu, R.: Institution-independent Model Theory. Studies in Universal Logic, vol. 2. Birkhäuser, Basel (2008)
6. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification, vol. 2. Springer, Heidelberg (1990)
7. Fiadeiro, J., Maibaum, T.: Temporal Theories as Modularisation Units for Concurrent System Specification. Formal Aspects of Computing 4(3) (1992)
8. Fiadeiro, J., Maibaum, T.: Describing, Structuring and Implementing Objects. In: de Bakker, J.W., Rozenberg, G., de Roever, W.-P. (eds.) REX 1990. LNCS, vol. 489, Springer, Heidelberg (1991)
9. Fiadeiro, J.: Categories for Software Engineering. Springer, Heidelberg (2004)
10. Garlan, D.: Software Architecture: A Roadmap. In: Filkenstein, A. (ed.) The Future of Software Engineering. ACM Press, New York (2000)
11. Goguen, J., Burstall, R.: Introducing Institutions. In: Hutchison, D., Shepherd, W.D., Mariani, J.A. (eds.) Local Area Networks: An Advanced Course. LNCS, vol. 184, Springer, Heidelberg (1985)
12. Goguen, J., Burstall, R.: Institutions: Abstract Model Theory for Specification and Programming. Journal of the ACM 39(1) (1992)
13. Goguen, J., Rosu, G.: Institution Morphisms. Formal Aspects of Computing 13(3-5) (2002)
14. Hirsch, D., Montanari, U.: Two Graph-Based Techniques for Software Architecture Reconfiguration. ENTCS, vol. 51. Elsevier, Amsterdam (2001)
15. Knapp, A., Marczynski, G., Wirsing, M., Zawlocki, A.: A Heterogeneous Approach to Service-Oriented Systems Specification. In: Proc. of the ACM Symposium on Applied Computing, SAC 2010. ACM Press, New York (2010)
16. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer, Heidelberg (1991)
17. Medvidovic, N.: ADLs and Dynamic Architecture Changes. In: Proc. of the 2nd. International Software Architecture Workshop ISAW-2 (1996)
18. Meseguer, J.: General Logics. In: Logic Colloquium 1987. North Holland, Amsterdam (1989)
19. Mossakowski, T., Maeder, C., Luttich, K.: The Heterogeneous Tool Set, Hets. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 519–522. Springer, Heidelberg (2007)
20. Pierce, B.: Basic Category Theory for Computer Scientists. The MIT Press, Cambridge (1991)

21. Sannella, D., Tarlecki, A.: Specifications in an Arbitrary Institution. Inf. Comput. 76(2/3) (1988)
22. Tarlecki, A.: Moving Between Logical Systems. In: Haveraaen, M., Dahl, O.-J., Owe, O. (eds.) Abstract Data Types 1995 and COMPASS 1995. LNCS, vol. 1130, Springer, Heidelberg (1996)
23. Tarlecki, A.: Toward Specifications for Reconfigurable Component Systems. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 24–28. Springer, Heidelberg (2007)
24. Tarlecki, A.: Towards Heterogeneous Specifications. In: Frontiers of Combining Systems. Studies in Logic and Computation, vol. 2. Research Studies Press, Hertfordshire (2000)
25. Tarlecki, A.: Abstract Specification Theory: An Overview. In: Proc. of the NATO Advanced Study Institute on Models, Algebras and Logic of Engineering Software. NATO Science Series. IOS Press, Amsterdam (2003)
26. Wermelinger, M., Lopes, A., Fiadeiro, J.: A Graph Based Architectural (Re)configuration Language. In: Proc. of the Joint European Software Engineering Conference the Symposium on the Foundations of Software Engineering, ESEC/FSE 2001. ACM Press, New York (2001)
27. Davies, J., Woodcock, J.: Using Z. Prentice-Hall, Englewood Cliffs (1996)

Characterizing Locality (Encapsulation) with Bisimulation

Pablo F. Castro[1] and Tom S.E. Maibaum[2]

[1] Universidad Nacional de Rio Cuarto, Departamento de Computación, Argentina
pcastro@dc.exa.unrc.edu.ar
[2] McMaster University, Department of Computing & Software, Hamilton, Canada
tom@maibaum.org

Abstract. In this paper we investigate formal mechanisms to allow designers to decompose specifications (stated in a given logic) into several components. The basic ideas come from [1] where some notions from category theory are used to put together logical specifications. In this setting the concept of locality allows designers to write separate specifications and then compose them. However, as the work of Fiadeiro and Maibaum [1] is stated in a linear temporal logic, we investigate how to extend these notions to a branching time logic, which can be used to specify systems where non-determinism is a relevant mechanism. Since we are interested in specifying and verifying fault-tolerant systems, we also introduce deontic operators in our logic, we have shown in [2] that deontic logic allows us to express notions such as ideal and abnormal behavior which are closely related to fault-tolerance.

Keywords: Fault-Tolerance, Software Specification, Software Verification, Deontic Logic, Component Based Design, Category Theory.

1 Introduction

In [2] we introduced a deontic logic which is designed for reasoning about fault-tolerant systems. The basic idea in this logic is to use deontic predicates (*permission*, *obligation*, *prohibition*, etc) to introduce in specifications the idea of normal (and abnormal) behavior. Deontic logics are those logics that originated with the use of deontic predicates for reasoning about legal or moral arguments [3]. However, in the past few decades, computer scientists have used deontic logics to reason about computing systems. Examples of applications can be found in artificial intelligence, agent systems, policy languages, databases, etc. (See [4] for a detailed list of applications of deontic logic in computer science.) In particular, we take the ideas presented in [5], where deontic predicates are used to distinguish between the description and the prescription of systems. The description of a system is given in a pre/post-condition style, while the prescriptions are given by means of deontic predicates establishing what the desirable behaviors of the system are. We think that this idea can be useful for fault-tolerance, where violations arise naturally when an abnormal behavior of the system occurs, and, therefore, some actions must be executed to recover the system from error states.

In this paper we introduce some modifications to the deontic logic presented in [2] with the aim of obtaining a more general framework where system specifications can be written in a modular way. For this purpose, we mainly follow the philosophy of [6], in the sense that a system is specified by putting together smaller specifications (by means of some categorical constructions [7]). The ideas presented below are also inspired by the logical frameworks presented in [8,1], where Goguen's ideas are applied to a temporal logic and, therefore, to specifications of concurrent systems and object oriented systems, respectively. We think that the prescriptions in specifications must respect the structure of the system, and therefore they must be local to components. For example, in [8] deontic operators are also used in specifications, but the deontic constructs there are used in a "global" way, in the sense that the prescriptions of one component may implicitly affect other parts of the system. This is undesirable when we want to perform modular reasoning about specifications. The logic introduced below maintains the locality of prescriptions, which we think is more useful for modular reasoning. Moreover, we introduce a modified notion of encapsulation in specifications, which allows us to define the notion of module or component. The novel part of these ideas is that we use a variation of bisimulation to introduce the idea that a component forms part of a system, and that its behavior is respected by the latter. Further, it should be noted that the non-determinism in a component is also respected by the system.

The paper is structured as follows. In the next section we introduce the basic definitions of the logic. In sections 3 and 4 we present the notion of component and the ideas used to put components together. We present an example of application in section 5. Basic notions of category theory are used throughout the following sections. The reader can consult the standard bibliography, for example [7].

2 Syntax and Semantics

In this section we present the syntax and semantics of the logic. The basic definitions are based on those given in [2]. We introduce some modifications to the definitions given in that paper to be able to capture the notions of component and violation. We use *vocabulary* (or *language*) to refer to a tuple $L = \langle \Delta_0, \Phi_0, V_0, I_0 \rangle$, where Δ_0 is a finite set of *primitive actions*: $a_1, ..., a_n$, which represent the possible actions of a part of the system and, perhaps, of its environment. Φ_0 is an enumerable set of propositional symbols denoted by $p_1, p_2, \ldots$. V_0 is a finite subset of $\mathcal{V}$, where $\mathcal{V} = \{\mathtt{v}_1, \mathtt{v}_2, \mathtt{v}_3, \ldots\}$ is an infinite, enumerable set of "violation" propositions. The indices in I_0 correspond to a stratification of the concept of norm, where the stratification corresponds to degrees of fault in the system being modeled. All these sets are mutually disjoint. Using the primitive actions we define the set Δ of actions as follows: if $a \in \Delta_0$, then a is an action. $\emptyset$ (the impossible action) and $\mathbf{U}$ (the non-deterministic choice of any primitive action) are actions. If α and β are actions, then $\alpha \sqcup \beta$ (the non-deterministic choice between α and β) is an action, and $\alpha \sqcap \beta$ (the parallel execution of α and β) is an action. If α is an action, then $\overline{\alpha}$ (the execution of an alternative action to α) is an action. No other expression is an action.

The formulae of this logic (denoted by Φ) are defined as follows. If $\varphi \in \Phi_0 \cup V_0$, then φ is a formula. If φ and ψ are formulae, then $\psi \rightarrow \psi, \neg\psi$ are formulae (these are the standard propositional connectives). If φ is a formula and α an action, then $[\alpha]\varphi$ (after executing α, φ is true) is a formula. If α is an action and $i \in I_0$, then $\mathsf{P}^i(\alpha)$ (α is allowed to be executed in any scenario at level i, this deontic operator is called strong permission) and $\mathsf{P}^i_{\mathsf{w}}(\alpha)$ (the action α is weakly allowed to be executed at level i) are formulae. If φ and ψ are formulae, then $\mathsf{EN}\varphi$ (in some path of execution in the next instant φ is true) is a formula. $\mathsf{A}(\varphi\ \mathcal{U}\ \psi)$ (in every path of execution φ is true until ψ becomes true) and $\mathsf{E}(\varphi\ \mathcal{U}\ \psi)$ (in some path of execution φ is true until ψ becomes true) are formulae. If α and β are actions and $S \subseteq \Delta_0$, then $\mathsf{Done}_S(\alpha)$ (the last action in S executed was α) and $\alpha =_{act} \beta$ (α and β when executed produce the same events) are formulae. B is a formula, it is true at beginning of time. Note that the temporal operators have the standard meaning in a branching temporal logic. Actions and action operators form a boolean algebra, we consider the laws of boolean algebra in our logical calculus. For any language $L = \langle \Phi_0, \Delta_0, V_0 \rangle$, we have a boolean algebra of action terms modulo the axioms of boolean algebra, we denote the members of this algebra by $\Delta_0 / =_{act}$. Since the number of primitive action symbols in L is finite, the boolean algebra of terms is atomic. The atoms are (members of the equivalence class of) monomials of the style $*a_1 \sqcap \ldots \sqcap *a_n$ where $*a_i$ is $\overline{a_i}$ or a_i and $a_1, \ldots, a_n$ are the primitive actions of L. Intuitively, a monomial of this kind indicates which actions are executed and which are not.

Some remarks are required about the deontic operators. The strong permission allows us to assert that a given action is allowed to be executed in any scenario. Contrast this to weak permission, which has to be used when we need to assert that an action is allowed to happen only in some scenarios. For instance, P(`wdraw`) (where `wdraw` is the action of withdrawing money from a cash machine) says that it is allowed to withdraw money from the machine. However, it may convenient to say that, the action `wdraw` can only be executed if the machine has enough money, and therefore we have to use a weak permission, i.e.: P_{w}(`wdraw`). Using permissions we can define other deontic operators. We say that an action is forbidden if it it is not allowed to be executed in any scenario, hence: $\mathsf{F}^i(\alpha) \stackrel{\mathsf{def}}{=} \neg\mathsf{P}^i_{\mathsf{w}}(\alpha)$. We say that an action is obliged to occur if it is allowed and any other action is forbidden to be executed, i.e.: $\mathsf{O}^i(\alpha) \stackrel{\mathsf{def}}{=} \mathsf{P}^i(\alpha) \wedge \neg\mathsf{P}^i_{\mathsf{w}}(\alpha)$. The index in the deontic predicates allows us to introduce different levels of normative restrictions. The levels are not necessarily related to each other, but a relation can be added by means of axiomatizations. These levels in the deontic operators allow us to distinguish between the norms of different components, e.g., avoiding that the obligation in a component to execute a given action affects the other components in the system (i.e., these components are not obliged to execute this action).

Now, we introduce the technical details. We follow the ideas of [8], where transitions can be produced by actions in the language or by external components (i.e., this is an *open system* approach in the sense that is given in [9]). Intuitively, each action produces a (finite) set of events during the execution of the system (the events that this action "observes" or "participates in"), and also there are other events produced by actions from other components or from the environment. We define the notion of semantic structure.

Definition 1 (models). *Given a language $L = \langle \Phi_0, \Delta_0, V_0, I_0 \rangle$, a L-Structure is a tuple: $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \{\mathcal{P}^i \mid i \in I_0\}, w_0 \rangle$ where:*

– *$\mathcal{W}$ is a set of worlds.*
– *$\mathcal{R}$ is an $\mathcal{E}$-labeled relation between worlds. We require that, if $(w, w', e) \in \mathcal{R}$ and $(w, w'', e) \in \mathcal{R}$, then $w' = w''$, i.e., $\mathcal{R}$ is functional.*
– *$\mathcal{E}$ is an infinite, enumerable non-empty set, of (names of) events.*
– *$\mathcal{I}$ is a function:*
  - *For every $p \in \Phi_0 : \mathcal{I}(p) \subseteq \mathcal{W}$*
  - *For every $\alpha \in \Delta_0 : \mathcal{I}(\alpha) \subseteq \mathcal{E}$, and $\mathcal{I}(\alpha)$ is finite.*

  In addition, the interpretation $\mathcal{I}$ has to satisfy the following properties:
  I.1 *For every $\alpha_i \in \Delta_0$: $|\mathcal{I}(\alpha_i) - \bigcup\{\mathcal{I}(\alpha_j) \mid \alpha_j \in (\Delta_0 - \{\alpha_i\})\}| \leq 1$.*
  I.2 *For every $e \in \mathcal{I}(a_1 \sqcup \ldots \sqcup a_n)$: if $e \in \mathcal{I}(\alpha_i) \cap \mathcal{I}(\alpha_j)$, where $\alpha_i \neq \alpha_j \in \Delta_0$, then: $\cap\{\mathcal{I}(\alpha_k) \mid \alpha_k \in \Delta_0 \wedge e \in \mathcal{I}(\alpha_k)\} = \{e\}$.*
– *w_0 is the initial state.*

Roughly speaking, the structure gives us a labeled transition system, whose labels are events, which are produced by some local action(s) or could also correspond to external events. Note that we have a set of events, but actions are only interpreted over finite subsets; condition **I.1** states that the isolated execution of an action produces at most a unique event. Condition **I.2** says that if we execute all the actions which produce a given event, then the execution of this maximal set of actions produce a unique event. These conditions ensure that every one-point set can be generated from the actions of the component; i.e., the labels in the transitions are uniquely determined by some parallel execution of actions in the component and environmental actions (see [2] for more details). We call *standard models* those structures where $\mathcal{E} = \bigcup_{\alpha \in \Delta_0} \mathcal{I}(\alpha)$, i.e., when we do not have "outside" events in the structure. Note that the semantics of the logic described in [2] is given only in terms of standard models.

We use maximal traces to give the semantics of the temporal operators. Given a L-structure $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \{\mathcal{P}^i \mid i \in I_0\}, w_0 \rangle$, an infinite trace (or path) is a sequence $\pi = w_0 \xrightarrow{e_0} w_1 \xrightarrow{e_1} w_2 \xrightarrow{e_2} \ldots$ (where each $w_i \xrightarrow{e_i} w_{i+1}$ is a labeled transition in M); we denote by $\pi^i = w_i \xrightarrow{e_i} w_{i+1} \xrightarrow{e_{i+1}} \ldots$ the subpath of π starting at position i. The notation $\pi_i = w_i$ is used to denote the i-th state in the path, and we write $\pi[i,j]$ (where $i \leq j$) for the subpath $w_i \xrightarrow{e_i} \ldots \xrightarrow{e_j} w_{j+1}$. $\pi(i)$ denotes the event e_i. Finally, given a finite path $\pi' = w'_0 \xrightarrow{e'_0} \ldots \xrightarrow{e'_n} w_{n+1}$, we say $\pi' \preceq \pi$ if π' is an initial subpath of π, that is: $w_i = w'_i$ and $e_i = e'_i$ for $0 \leq i \leq n$, and we denote by $\prec$ the strict version of $\preceq$. We denote by $\#\pi$ the length of the trace π; if it is infinite we abuse notation and say $\#\pi = \infty$. The set of sequences in M starting in w_0 is denoted by $\Sigma(w_0)$, and the set of maximal sequences starting at w_0 is denoted by $\Sigma^*(w_0)$.

Since, in a trace, we have events that do not belong to the actual component, we need to distinguish between those events generated by the component being specified and those which are from the environment. Given language L, a L-structure M and a maximal path π in M, we define the set:

$$\mathrm{Loc}_L(\pi) = \{i \mid \pi(i-1) \in \mathcal{I}(a_1 \sqcup \ldots \sqcup a_n)\} \cup \{0\}$$

(where $a_1, \ldots, a_n$ are all the primitive actions of L), i.e., this set contains all the positions of π where events occur that are observed by some action in L. Obviously, this set is totally ordered by the usual relationship $\leq$. Also, we consider a restricted version of this set; given a set $\{a_1, \ldots, a_m\} \subseteq \Delta_0$, we define:

$$Loc_{\{a_1,\ldots,a_m\}}(\pi) = \{i \mid \pi(i-1) \in \mathcal{I}(a_1 \sqcup \ldots \sqcup a_m)\}.$$

That is, in this case we restrict ourselves to the actions belonging to $\{a_1, \ldots, a_m\}$; this restricted set of positions is useful when we need to reason about a part of a system. In the following, given a set S of naturals, we denote by $\min_p(S)$ the minimum element in S which satisfies the predicate p, and similarly for $\max_p(S)$. Using these concepts, we define the relationship $\vDash_L$ between structures and formulae of a given language L. Note that we introduce the definition the semantics in a similar way to that in [2], but taking into account the separation between local and external events.

Definition 2. *Given a trace $\pi = w_0 \xrightarrow{e_0} w_1 \xrightarrow{e_1} w_2 \xrightarrow{e_2} \ldots \in \Sigma^*(w_0)$, we define the relation $\vDash_L$ as follows:*

- $\pi, i, M \vDash_L \mathsf{B} \overset{\mathsf{def}}{\iff} i = 0.$
- *If $p_j \in \Phi_0 \cup V_0$, then $\pi, i, M \vDash_L p_j \overset{\mathsf{def}}{\iff} \pi_i \in \mathcal{I}(p_j)$.*
- $\pi, i, M \vDash_L \alpha =_{act} \beta \overset{\mathsf{def}}{\iff} \mathcal{I}(\alpha) = \mathcal{I}(\beta).$
- $\pi, i, M \vDash_L \mathsf{P}^i(\alpha) \overset{\mathsf{def}}{\iff} \forall e \in \mathcal{I}(\alpha) : \mathcal{P}^i(w, e).$
- $\pi, i, M \vDash_L \mathsf{P}^i_{\mathsf{w}}(\alpha) \overset{\mathsf{def}}{\iff} \exists e \in \mathcal{I}(\alpha) : \mathcal{P}^i(w, e).$
- $\pi, i, M \vDash_L \neg\varphi \overset{\mathsf{def}}{\iff}$ *not* $\pi, i, M \vDash_L \varphi$.
- $\pi, i, M \vDash_L \varphi_1 \rightarrow \varphi_2 \overset{\mathsf{def}}{\iff}$ *either not* $\pi, i, M \vDash_L \varphi_1$ *or* $\pi, i, M \vDash_L \varphi_2$.
- $\pi, i, M \vDash_L \mathsf{Done}_S(\alpha) \overset{\mathsf{def}}{\iff} \exists j : j = max_{<i}(Loc_S(\pi)) \wedge e_j \in \mathcal{I}(\alpha).$
- $\pi, i, M \vDash_L [\alpha]\varphi \overset{\mathsf{def}}{\iff} \forall \pi' = s'_0 \xrightarrow{e'_0} s'_1 \xrightarrow{e'_1} \ldots \in \Sigma^*(w)$ *such that* $\pi[0, i] \prec \pi'$, *if* $j = min_{>i}(Loc(\pi'))$, *and if* $e'_j \in \mathcal{I}(\alpha)$, *then* $\pi', j, M \vDash_L \varphi$.
- $\pi, i, M \vDash_L \mathsf{AN}\varphi \overset{\mathsf{def}}{\iff}$ *if* $i = \#\pi$, *then* $\pi, i, M \vDash \varphi$. *If* $i \neq \#\pi$, *then* $\forall \pi' \in \Sigma^*(w) : \pi[0..i] \prec \pi'$:, *if* $j = min_{>i}(Loc(\pi'))$, *then* $\pi', j, M \vDash \varphi$.
- $\pi, i, M \vDash_L \mathsf{A}(\varphi_1\ \mathcal{U}\ \varphi_2) \overset{\mathsf{def}}{\iff}$ *if* $i = \#\pi$, *then* $\pi, i, M \vDash \varphi_2$. *If* $i \neq \#\pi$, *then* $\forall \pi' \in \Sigma^*(w) : \pi[0..i] \prec \pi'$ *we have that* $\exists j \in Loc((\pi')^i) : \pi', j, M \vDash \varphi_2$ *and* $\forall i \leq k \leq j : k \in Loc((\pi')^i)$, *then* $\pi', k, M \vDash \varphi_1$.
- $\pi, i, M \vDash_L \mathsf{E}(\varphi_1\ \mathcal{U}\ \varphi_2) \overset{\mathsf{def}}{\iff}$ *if* $i = \#\pi$, *then* $\pi, i, M \vDash \varphi_2$. *If* $i \neq \#\pi$, *then* $\exists \pi' \in \Sigma^*(w) : \pi[0..i] \prec \pi'$ *such that* $\exists j \in Loc((\pi')^i) : \pi', j, M \vDash \varphi_2$ *and* $\forall i \leq k \leq j : k \in Loc((\pi')^i)$, *then* $\pi', k, M \vDash \varphi_1$.

We say that $M \vDash_L \varphi$ when $\pi, i, M \vDash \varphi$ for every path π and instant i. And we say that $\vDash_L \varphi$ when $M \vDash \varphi$ for every model M. Note that we use the set $Loc(\ldots)$ to observe the events that are only produced by local actions. We can think of the propositional variables in L as local variables, which cannot be changed by other components, i.e., we must require (as is done in [8,1]) that external events do not produce changes in local variables. In [1] the notion of a *locus* trace is introduced to reflect this property in the logic; a locus (trace) is one in which the external events do not affect the state of local variables. However, the logic used in that work is a linear temporal logic, and this implies that here we cannot restrict only to traces to express this requirement, since we have a branching temporal logic. In the following we take further the ideas introduced in [1] and we define *locus models* which have the property of generating locus traces.

3 Locus Models

Given a structure M over a language L, we say that an event e is *non-local* if it does not belong to the interpretation of any action of the language; otherwise, we say that it is a *local event*. We say $w \overset{\epsilon}{\Rightarrow} w'$, if there exists a path $w \xrightarrow{e_0} w_1 \xrightarrow{e_1} w_2 \xrightarrow{e_2} \ldots \xrightarrow{e_n} w_n$ in M, such that e_i is non-local for every $0 \leq i \leq n$. We say that $w \overset{\infty}{\Rightarrow}$ when there is an infinite path from w: $w \xrightarrow{e_0} w_1 \xrightarrow{e_1} \ldots$, such that every e_i is non-local. Furthermore, we say $w \overset{e}{\Rightarrow} w'$ (where e is local) if $w \overset{\epsilon}{\Rightarrow} w''$ and $w'' \xrightarrow{e} w'$. Given two L-structures $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \{\mathcal{P}^i \mid i \in I_0\}, w_0 \rangle$ and $M' = \langle \mathcal{W}', \mathcal{R}', \mathcal{E}', \mathcal{I}', \{\mathcal{P}'^i \mid i \in I_0\}, w'_0 \rangle$, such that $\mathcal{I}(\alpha) = \mathcal{I}'(\alpha)$ for any α, we say that a relationship $Z \subseteq \mathcal{W} \times \mathcal{W}'$ is a *local bisimulation* between M and M' iff:

- If wZv, then $L(w) = L(v)$.
- If wZv, and $w \overset{\infty}{\Rightarrow}$, then either $v \overset{\infty}{\Rightarrow}$ or there is a v' such that $v \overset{\epsilon}{\Rightarrow} v'$ and v' has no successors by $\rightarrow$ in M.
- if wZv and $w \xrightarrow{e} w'$. then $w'Zv$ if e is non-local. Otherwise we have some v' such that $v \xrightarrow{e} v'$ and $w'Zv'$.
- $Z^{\smile}$ also satisfies the above conditions (where $Z^{\smile}$ is the converse of Z).

Here $L(v)$ denotes the set of all the state formulae (primitive propositions, deontic predicates and equations) true at state v. In branching bisimulation (as defined in [10]), we can "jump" through non-local events; however, here we require a stronger condition: we can move through non-local events, but, if we have the possibility of executing a local event, we must have the same possibility in the related state. We see later on that this notion of bisimulation induces useful properties on the models and that we can characterize this notion in an axiomatic way.

We say that two models M and M' are (locally) *bisimilar* iff $w_0 Z w'_0$ (where w_0 and w'_0 are the corresponding initial states) for some local bisimulation Z; we denote this situation by $M \sim_Z M'$. We prove later on that two bisimilar models are indistinguishable by our logic. In [10], it is shown that CTL^*-X (CTL^* without the next operator) cannot distinguish between Kripke structures which are DBSB (divergent blind stuttering) bisimilar; however, in the semantics of the temporal logic considered in that work, there are no labels on the transitions and, therefore, the next operator is problematic since it is interpreted as a global next operator. Here we can take advantage of the fact that we have the events as labels of transitions, and, therefore, we can distinguish between local and non-local transitions. Furthermore, note that our next operator is a local one (although this implies some subtle technical points when it comes to defining the composition of components, see below).

Using bisimilarity, we define the notion of a *locus* model (following the terminology of [1] where locus models are introduced in a linear temporal logic).

Definition 3. *We say that a structure M is a locus iff there is a local bisimulation between M and a standard model M'.*

We use this notion of bisimulation to formalize the notion of locus structure that, as shown later on, will be essential in defining composition of modules (or components). Roughly speaking, locus models are those which have a behavior which is, essentially, the same as that of a standard model. Hence, the usual notion of encapsulation, as informally understood in software engineering, applies to our concept of component: only local actions can modify the values of local variables.

Below we present the main results about bisimulations; because of space restrictions, we do not show the technical proofs in this paper; the proofs can be consulted in [11]. The following theorem says that bisimilar models satisfy the same predicates.

Theorem 1. *If $M \sim_Z M'$, then $M \vDash \varphi$ iff $M' \vDash \varphi$.*

In section 2 we introduced non-standard models (i.e., those models which have "external" events). However, not all non-standard models are useful; we want that the external events preserve local variables, that is, the events not generated by any of the actions in the component have to be silent, in some sense. In [1], with the same purpose in mind, the notion of locus trace is introduced. A *locus* trace is one in which, after executing a non-local event, the local variables retain their value. However, since we have a branching time logic and a modal logic, here it is not enough to just put restrictions on traces. We need to take into account the branching occurring in the semantic structures, i.e., we need a more general notion of locus model.

Roughly speaking, locus models are those which are locally bisimilar to a standard model. In some sense, this definition characterizes those models which behave as standard models, where the external actions are silent with respect to local attributes.

Definition 4. *Given a language L, we say that a L-structure M' is a locus iff there is a standard model M such that $M \sim_Z M'$ for some local bisimulation Z.*

Using the result presented above about local bisimulation, we get that locus structures do not add anything new to the logic (w.r.t. formula validity).

Theorem 2. *If M is a locus structure, then $M \vDash \varphi$ iff there is some standard structure M' such that $M' \vDash \varphi$.*

Summarizing, nothing is gained or lost in using the locus models of a given language. However, we want to use these kinds of models over a wider notion of logical system; we shall consider several languages and translations between them, and therefore we need to have a notion of model which agrees with the locality properties of a language when we embed this language in another. First, let us define what a translation between two languages is.

Definition 5. *A translation τ between two languages $L = \langle \Delta_0, \Phi_0, V_0, I_0 \rangle$ and $L' = \langle \Delta'_0, \Phi'_0, V'_0, I'_0 \rangle$ is given by: (i) A mapping $f : \Delta_0 \to \Delta'_0$ between the actions of L and the actions of L'; (ii) A mapping $g : \Phi_0 \to \Phi'_0$ between the propositions of L and the predicates of L'; (iii) A mapping $h : V_0 \to V'_0$, between the violations of L and the violations of L'; (iv) A mapping $i : I_0 \to I'_0$.*

For the sake of simplicity, we denote the application of any of these functions using the name of the mapping, e.g., instead of writing $f(a_i)$ we write $\tau(a_i)$. In [11] we presented a sound axiomatic system for this semantics. Because we want to keep this presentation brief we do not introduce all the axioms and technical details here, but we say that $\vdash^L \varphi$ if the formula φ can be proven from this axiomatic system in the language L.

The collection of all the languages and all the translations between them forms the category **Sign**. It is straightforward to see that it is really a category: identity functions define identity arrows, and composition of functions gives us the composition of translations (which straightforwardly satisfy associativity). Now, given a translation, we can extend this translation to formulae (actually we can describe a grammar functor which reflects these facts, as done in Institutions [12] or π-Institutions [13]). Given a translation $\tau : L \to L'$ as explained above, we extend τ to a mapping between the formulae of L and those of L', as follows. First, we need to define how the translation behaves with respect to action terms: $\tau(\alpha \sqcup \beta) = \tau(\alpha) \sqcup \tau(\beta)$, $\tau(\emptyset) = \emptyset$, $\tau(\alpha \sqcap \beta) = \tau(\alpha) \sqcap \tau(\beta)$, $\tau(\overline{\alpha}) = \tau(\mathbf{U}) \sqcap \overline{\tau(\alpha)}$ and $\tau(\mathbf{U}) = \tau(a_1) \sqcup \ldots \sqcup \tau(a_n)$ (where $\Delta_0 = \{a_1, \ldots, a_n\}$). Note that the complement is translated as a relative complement, and the universal action is translated as the non-deterministic choice of all the actions of the original component (which is different from the universal action in the target language). It is important to stress that some extra axioms must be added to the axiomatic system to deal with the fact that the actions are interpreted as being relative to a certain universe. Now, the extension to formulae is as follows:

– $\tau([\alpha]\varphi) = [\tau(\alpha)]\tau(\varphi)$, $\tau(\neg\varphi) = \neg(\tau(\varphi))$, $\tau(\varphi \to \psi) = \tau(\varphi) \to \tau(\psi)$
– $\tau(\mathsf{AN}\varphi) = (\langle \tau(\mathbf{U}) \rangle \top \to \mathsf{AN}(\mathsf{Done}(\tau(\mathbf{U}))) \to \tau(\varphi))) \vee ([\tau(\mathbf{U})]\bot \to \tau(\varphi))$
– $\tau(\mathsf{A}(\varphi\, \mathcal{U}\, \psi)) = \mathsf{A}(\tau(\varphi)\, \mathcal{U}\, \tau(\psi))$
– $\tau(\mathsf{E}(\varphi\, \mathcal{U}\, \psi)) = \mathsf{E}(\tau(\varphi)\, \mathcal{U}\, \tau(\psi))$
– $\tau(\mathsf{Done}_S(\alpha)) = \mathsf{Done}_{\tau(S)}(\tau(\alpha))$, where $\tau(S) = \{\tau(a_i) \mid a_i \in S\}$

τ is a function which preserves logical symbols. In other words, using translations between signatures, we can define morphisms between formulae, and therefore we can define interpretations between theories (in the standard sense). We deal with this issue in the next section.

Given a translation $\tau : L \to L'$ and given a L'-structure M, it is straightforward to define the restriction of $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \{\mathcal{P}^i \mid i \in I_0\}\rangle$ with respect to τ (or its reduct as it is called in model theory), as follows:

Definition 6. *Given a translation $\tau : L \to L'$ and a L'-structure M we can define a L-structure $M|_\tau$ as follows:*

- $\mathcal{W}|_\tau = \mathcal{W}$.
- $\mathcal{E}|_\tau = \mathcal{E} - \{e \mid e \in \mathcal{I}(\tau(\mathbf{U})) \cap \mathcal{I}'(\Delta'_0 - \tau(\Delta_0))\}$*, where for any set of primitive actions S we define: $\mathcal{I}(S) = \bigcup\{\mathcal{I}(s) \mid s \in S\}$ and $\tau(S) = \bigcup\{\tau(s) \mid s \in S\}$.*
- $\mathcal{I}|_\tau(a_i) = \{e \in \mathcal{I}(a) \mid e \in \mathcal{E}|_\tau\}$*, for every $a_i \in \Delta'_0$.*
- $\mathcal{I}|_\tau(p_i) = \mathcal{I}(\tau(p_i))$*, for every $p_i \in \Phi'_0$.*
- $\mathcal{R}|_\tau = \{w \xrightarrow{e} w' \in \mathcal{R} \mid e \in \mathcal{E}|_\tau\}$.
- $\mathcal{P}^i|_\tau(w, e) \Leftrightarrow P^{\tau(i)}(w, e)$.
- $w_0|_\tau = w_0$.

It is worth noting that the restriction of a standard structure of L' can be a non-standard structure of L. Note also that we take out of the model those events which belong to translated actions and actions outside of the translation (see item 2 of definition 6), i.e., we only keep those events which are obtained by executing only actions of L or those which are obtained by executing actions outside of L. Some restrictions added below ensure that no important property of the original model is lost when we take its reduct.

Translations between languages and restrictions between models define a functor which is used in Institutions [12] to define logical systems. An important problem with restrictions is that a restriction of a given structure could be a structure which is not a locus, i.e., the obtained semantic entity violates the notion of locality as explained above. Furthermore, perhaps the reduct of a model loses some important properties. For this reason, we introduce the concept of τ-locus structures. We define some requirements on translations; given a translation $\tau : L \to L'$, consider the following set of formulae of the form:

- $\langle \tau(\gamma) \rangle \top \to \langle \tau(\gamma) \sqcap \overline{a_1} \sqcap \ldots \sqcap \overline{a_n} \rangle \top$, where γ is an atom of the boolean term algebra Δ_0/Φ_{BA}, and $a_1, \ldots, a_n \in \Delta'_0 - \tau(\Delta_0)$.

These formulae say that the execution of the actions of L when translated to L' are not dependent on any action of L'; we can think of this as an independence requirement, i.e., the actions of L when translated to L' keep their independence. This is an important modularity notion. In practice, this can be ensured by implementing the two components (which these languages describe) in different processes. We denote this set of formulae by $\mathsf{ind}(\tau)$. Another requirement (which is related to independence) is that the new actions in Δ'_0 (those which are not translations of any action in L) do not add new non-determinism to the translated actions. This fact can be expressed by the set of formulae of the following form:

– $\langle\tau(\gamma)\rangle\tau(\varphi) \to [\tau(\gamma)]\tau(\varphi)$, for every atom γ of the boolean algebra of terms obtained from L, and formula φ of L.

For a given translation $\tau : L \to L'$, we denote this set of formulae by $\mathsf{atom}(\tau)$, since they reflect the fact that the atomicity of the actions in L is preserved by translation.

Definition 7. *Given a translation $\tau : L \to L'$ and a L'-structure M, we say that M is a τ-locus iff $M \vDash \mathsf{ind}(\tau)$, $M \vDash \mathsf{atom}(\tau)$ and $M|_\tau$ is a locus structure for L.*

That is, a locus structure with respect to a translation τ is a structure which respects the locality and independence of L. We have obtained a semantical characterization of structures which respect the local behavior of a language with respect to a given translation. Because we wish to use deductive systems to prove properties over a specification, it is important to obtain some axiomatic way of characterizing this class of structures. For a given translation $\tau : L \to L'$, consider the following (recursive) set of formulae: $\{\tau(\varphi) \to [\overline{\tau(\mathbf{U})}]\tau(\varphi) \mid \varphi \in \Phi'\}$. Roughly speaking, this set of axiom schemes says that if an action of an external component is executed, then the local state of the current module is preserved. Note that, in [1], a similar set of axioms is proposed, although in that case it is a finite set, since that work uses a linear temporal logic, and therefore preserving only the propositions is enough for obtaining a satisfactory notion of locality. However, we need other axioms to express the property that when we embed a module inside another part of the system, we want to ensure that the behavior of the smaller module is preserved, in the following sense: we can introduce external events in some way in a given trace but we do not want that these external events add divergences that were not in the original trace. The following axiom does this: $\langle\tau(\mathbf{U})\rangle\top \to \mathsf{AFDone}(\tau(\mathbf{U}))$. This axiom expresses one of the conditions of local bisimulation, namely a trace cannot diverge by non-local events unless the component cannot execute any local action. It is worth noting that, if a local action is enabled in some state, then after executing a non-local action it will continue being enabled (as a consequence of the axiomatic schema described above), i.e., we require a fair scheduling of components, one which will not always neglect a component wishing to execute some of its actions.

Given a translation $\tau : L \to L'$, we denote this set of axioms, together with the axiomatic schema described above and the formulae $\mathsf{ind}(\tau)$ and $\mathsf{atom}(\tau)$, by $Loc(\tau)$. A nice property is that this set of formulae characterizes the L'-structures which are τ-loci.

Theorem 3. *Given a translation $\tau : L \to L'$, then a L'-structure is a τ-locus iff $M \vDash Loc(\tau)$.*

In the following, by $\Gamma \vdash^L \varphi$ and $\vdash^L_S \varphi$ we denote two different situations. The first can be thought of as a "local" deduction relationship. This relationship holds when we have a proof, in the standard sense, of φ where some members of Γ may appear, but the only rule that we can apply to them is *modus ponens*. Alternatively, $\vdash^L_S \varphi$ says that, if we extend our axiomatic system with the formulae of S, then we can prove φ.

An important property is that the deductive machinery obtained by adding the locality axioms preserves translations of properties.

Theorem 4. *Given a translation $\tau : L \to L'$, if $\vdash^L \varphi$, then $\vdash^L_{Loc(\tau)} \tau(\varphi)$.*

4 Defining Components

A component is a piece of specification which is made up of a language, a finite set of axioms, and a set of additional axioms which formalize implicit assumptions on the components (e.g., locality axioms). These implicit axioms are not intended to be defined by a designer; instead, they are automatically obtained from the structure of our system (using the relationships between the different components).

Definition 8. *A component is a tuple $\langle L, A, S \rangle$ where: L is a language, as described in earlier sections, A is a finite set of axioms (the properties specified by the designers) and S is a set of axioms (the system axioms expressing implicit restrictions like locality).*

Given a component $C = \langle L, A, S \rangle$, we denote by $\vdash_C \varphi$ the assertion $\vdash^L_{A,S} \varphi$. A mapping between two components is basically an interpretation between the theory presentations that define them.

Definition 9. *A mapping $\tau : C \to C'$ between two components $C = \langle L, A, S \rangle$ and $C' = \langle L', A', S' \rangle$ is a translation $\tau : L \to L'$ such that: (i) $\vdash_{C'} \tau(\varphi)$, for every $\varphi \in A \cup S$. (ii) $\vdash_{C'} Loc(\tau)$.*

It is worth noting that we require that the locality axioms must be theorems in the target component to ensure that the properties of the smaller component are preserved. This is expressed by the following corollary.

Theorem 5. *If $\tau : C \to C'$ is a mapping between components C and C', then: $\vdash_C \varphi \Rightarrow \vdash_{C'} \tau(\varphi)$.*

Now that we have a notion of component, we need to have some way to put components together. We follow Goguen's ideas [6], where concepts coming from category theory are used to put together components of a specification. The same ideas are used in [8,1], where temporal theories are used for specifying pieces of concurrent programs, and translations between them are used for specifying the relationships between these components. The idea then is to define a category where the objects are components (specifications) and the arrows are translations between them; therefore, putting together components is achieved by using the construction of colimits. Of course, some prerequisites are required. Firstly, the category of components has to be finitely cocomplete and, secondly, the notion of deduction has to be preserved by translations (which is exactly what we proved above).

First, recall that the collection of all the languages and all the translations between them form the category **Sign**. Components and mappings between them also constitute a category.

Theorem 6. *The collection of all components and all the arrows between them form the category* ***Comp****.*

The initial element of this category is the component with an empty language. Note that since the category of signatures is finitely cocomplete (its elements are just tuples of finite sets), the category of components is also finitely cocomplete; the forgetful functor from components to signatures reflects finite colimits (as shown for different logics in [12,13]).

Theorem 7. *The category* ***Comp*** *is finitely cocomplete.*

5 An Example

We present a simple example to illustrate these notions. We revisit the example that we presented in [2]. This example is a variation of Dijkstra's dining philosophers. We add the possibility that philosophers get sick and therefore they may have to go to the bathroom. The new scenario occurs when a philosopher takes some forks with him. (Obviously the worst scenario is when a philosopher takes with him two forks.) Here we follow the main ideas introduced in [1] to modularize the design of the standard version of Dijkstra's philosophers; note that no deontic operators are used in the referenced work. We introduce some notation to reduce the number of axioms in the specification shown below. The expression $\alpha \rightsquigarrow \varphi$ (where α is an action and φ a formula) denotes the following formula: $(\neg\varphi \rightarrow [\overline{\alpha}]\neg\varphi) \wedge ([\alpha]\varphi)$. Intuitively, this formula says that the action α is the only one which sets φ to true. Further notation can be introduced to obtain a higher level specification language, we leave this for further work.

First, let us consider the specification of a fork. The language of a fork has the following actions: $\Delta_0 = \{\texttt{l.up}, \texttt{l.down}, \texttt{r.up}, \texttt{r.down}\}$ and the following predicates: $\{\texttt{l.up?}, \texttt{r.up?}\}$. Intuitively, we have two ports by means of which we can use the forks; one is for the left philosopher and the other one is for the right philosopher. Note that this implies that the philosophers do not coordinate directly via any action (also note that these actions are mutually disjoint). The axioms of the fork are shown in figure 1. As explained above, a fork can be held onto by the

$\mathbf{f_1}.\mathsf{B} \rightarrow \neg\texttt{l.up?} \wedge \neg\texttt{r.up?}$	$\mathbf{f_4}.\neg(\texttt{l.up?} \wedge \texttt{r.up?})$	$\mathbf{f_7}.\texttt{l.up?} \rightarrow \langle\texttt{l.down}\rangle\top$
$\mathbf{f_2}.(\texttt{l.up} \rightsquigarrow \texttt{l.up?}) \wedge (\texttt{l.down} \rightsquigarrow \neg\texttt{l.up?})$	$\mathbf{f_5}.\neg\texttt{l.up?} \rightarrow [\texttt{l.down}]\bot$	$\mathbf{f_8}.\texttt{r.up?} \rightarrow \langle\texttt{r.down}\rangle\top$
$\mathbf{f_3}.(\texttt{r.up} \rightsquigarrow \texttt{r.up?}) \wedge (\texttt{r.down} \rightsquigarrow \neg\texttt{r.up?})$	$\mathbf{f_6}.\neg\texttt{r.up?} \rightarrow [\texttt{r.down}]\bot$	

Fig. 1. **XFork** specification

philosopher on the left or by the philosopher on the right. Therefore, we have two actions that reflect this action: `l.up` and `r.up`. Obviously they are disjoint (as stated by axiom $\mathbf{f_4}$), meaning that only one of the philosophers can be holding onto the fork.

The specification for a philosopher is shown in figure 2. The actions of the specification are the following: $\{\mathtt{getthk}, \mathtt{getbad}, \mathtt{gethungry}, \mathtt{up_L}, \mathtt{up_R}, \mathtt{down_L}, \mathtt{down_R}\}$. The action $\mathtt{getthk}$ indicates when the philosopher goes to the *thinking* state, $\mathtt{getbad}$ takes the philosopher to the *sick* state. The action $\mathtt{gethungry}$ takes a philosopher from the *thinking* state to the *hungry* state. The actions $\mathtt{up_L}$ and $\mathtt{up_R}$ are used for the philosopher to take the left or right fork, respectively. The predicates of the component are the following: $\{\mathtt{has_L}, \mathtt{has_R}, \mathtt{thk}, \mathtt{eating}, \mathtt{hungry}, \mathtt{bad}, \mathtt{has_L}, \mathtt{has_R}\}$. In addition, we have two violations $\{\mathtt{v}_1, \mathtt{v}_2\}$. The set of axioms of this specification is shown in figure 2. We consider the following predicate on the system axioms:

$\mathbf{p_1}: \mathsf{B} \rightarrow \neg\mathtt{v}_1 \wedge \neg\mathtt{v}_2 \wedge \mathtt{thk} \wedge \neg\mathtt{hungry} \wedge \neg\mathtt{bath}$

$\mathbf{p_2}: \mathtt{thk} \veebar \mathtt{hungry} \veebar \mathtt{eating} \veebar \mathtt{bad}$

$\mathbf{p_3}: \mathtt{eating} \leftrightarrow \mathtt{has_L} \wedge \mathtt{has_R} \wedge \neg\mathtt{bad}$

$\mathbf{p_4}: \neg\mathtt{hungry} \rightarrow \mathsf{AF}\mathtt{hungry}$

$\mathbf{p_5}: \neg\mathtt{eating} \rightarrow \mathsf{P}^1(\mathbf{U})$

$\mathbf{p_6}: \mathtt{eating} \rightarrow \mathsf{O}^1(\mathtt{down_L} \sqcap \mathtt{down_R})$

$\mathbf{p_7}: \mathsf{F}^1(\mathtt{down_L}) \rightarrow \overline{[\mathtt{down_L}]}\mathtt{v}_1$

$\mathbf{p_8}: \mathsf{F}^1(\mathtt{down_R}) \rightarrow \overline{[\mathtt{down_R}]}\mathtt{v}_2$

$\mathbf{p_9}: \mathtt{v}_1 \rightarrow \overline{[\mathtt{down_L}]}\mathtt{v}_1 \wedge \mathtt{v}_2 \rightarrow \overline{[\mathtt{down_R}]}\mathtt{v}_2$

$\mathbf{p_{10}}: \mathtt{v}_1 \rightarrow [\mathtt{down_L}]\neg\mathtt{v}_1 \wedge \mathtt{v}_2 \rightarrow [\mathtt{down_R}]\neg\mathtt{v}_2$

$\mathbf{p_{11}}: (\mathtt{getthk} \rightsquigarrow \mathtt{thk}) \wedge (\mathtt{getbad} \rightsquigarrow \mathtt{bath}) \wedge(\mathtt{gethungry} \rightsquigarrow \mathtt{hungry})$

$\mathbf{p_{12}}: \mathtt{thk} \rightarrow \mathtt{down_L} \wedge \mathtt{down_R}$

$\mathbf{p_{13}}: \mathtt{hungry} \rightarrow \mathtt{down_L} \wedge \mathtt{down_R}$

$\mathbf{p_{14}}: \mathtt{hungry} \wedge \langle\mathtt{up_L} \sqcup \mathtt{up_R}\rangle\top \rightarrow \mathsf{AN}\mathtt{eating} \vee \mathsf{AN}\mathtt{hungry}$

$\mathbf{p_{15}}: \mathtt{bad} \rightarrow \overline{[\mathtt{getthk}]}\mathtt{bad}$

$\mathbf{p_{16}}: \mathtt{eating} \rightarrow \mathsf{AN}(\mathtt{thk} \vee \mathtt{bad})$

Fig. 2. Phil specification

$\neg\mathtt{v}_i \wedge \mathsf{P}^1(\alpha) \rightarrow [\alpha]\neg\mathtt{v}_i$, for every i. This predicate basically states that allowed actions do not introduce new violations. Most of the axioms a self-explanatory, we discuss the remaining axioms. Axiom $\mathbf{P_4}$ says that a philosopher who is thinking will become hungry in the future; axiom $\mathbf{P_5}$ states that, when the philosopher is not eating, then everything is allowed. Axioms $\mathbf{p_7}$-$\mathbf{p_{10}}$ specify how the violations occur in a given execution of this specification and which are the recovery actions. Note that, in axiom $\mathbf{P_6}$, we say that, if a philosopher is eating, then it will be obliged to return both forks. We simplify the problem by requiring that philosophers can only eat for a unit of time (axiom $\mathbf{p_{16}}$).

Suppose that we want to obtain the specification of two philosophers sharing two forks. We need to define some way of connecting the different components. With this goal in mind, we define a component **Chan** which only has actions $\{\mathtt{port}_1, \mathtt{port}_2\}$ with no predicates and no violations. Using channels, we can connect the forks with the philosopher taking the colimit of the diagram shown in figure 3. The components $\mathbf{XFork^1}$ and $\mathbf{XFork^2}$ are "instances" of the specification **XFork** (i.e., they are obtained from **XFork** by renaming the symbols with the subindex i), and $\mathbf{Chan^1}, \mathbf{Chan^2}, \mathbf{Chan^3}, \mathbf{Chan^4}$ are "instances" of **Chan**. Here $l_1 : \mathbf{Chan^1} \rightarrow \mathbf{XFork^1}$ maps $\mathtt{port}^1_1 \mapsto \mathtt{lup}$ and $\mathtt{port}^1_2 \mapsto \mathtt{ldown}$, whereas $o_1 : \mathbf{Chan^1} \rightarrow \mathbf{Phil^1}$ maps $\mathtt{port}^1_1 \mapsto \mathtt{up_L}$ and $\mathtt{port}^1_2 \mapsto \mathtt{down_L}$ and similarly for the other mappings. In other words, these morphisms connect the right and the left fork with the philosopher. The colimit of this diagram gives us the final design (say **2Phils**). Note that this colimit produces the corresponding specification with all the needed renaming of clashing symbols. Also, the colimit "produces" a new level of permission, obligation or prohibition for each component, avoiding in this way the introduction of global norms which may result in inconsistencies in the final design.

$$\begin{tikzcd}
\mathbf{Channel^1} \arrow[r, "l_1"] \arrow[d, "o_1"'] & \mathbf{XFork^1} & \mathbf{Channel^3} \arrow[l, "r_2"'] \arrow[d, "o_2"] \\
\mathbf{Phil^1} & & \mathbf{Phil^2} \\
\mathbf{Channel^2} \arrow[r, "r_1"'] \arrow[u, "o_3"] & \mathbf{XFork^2} & \mathbf{Channel^4} \arrow[l, "l_2"] \arrow[u, "o_4"']
\end{tikzcd}$$

Fig. 3. Two philosophers eating

It is interesting to remark that the two instances of **Phil** do not coordinate via any action (both coordinate with **XFork**, but using different channels); this implies that the actions of $\mathbf{Phil^1}$ do not introduce violations in the states of $\mathbf{Phil^2}$, and viceversa. This is an important property because we can guarantee that the recovery actions of one component do not introduce violations in other components. Note also that, because of the formula introduced in the system axioms, if we coordinate the components via allowed actions, we also obtain that the locality of violations is preserved (i.e., the actions of one component do not add more violations to the states of other components). These kinds of independence properties seem very important when reasoning about composition of modules which contain violations. Roughly speaking, this enables a compositional reasoning about fault-tolerance. We have not shown in detail these properties, but the interested reader can find them in [11].

6 Further Remarks

In this paper we delineate a basic framework to modularize deontic specifications. The main idea is to use a notion of bisimulation to capture the concept of encapsulation or locality, which is, obviously, related to the concept of module or component. In contrast to the work of Fiadeiro and Maibaum [1], where a linear time logic is used, the main formalism used in this paper is a branching time logic, in which non-determinism is naturally reflected. In addition, we provide deontic predicates which can be used to specify ideal behavior of systems, and therefore to model some concepts related to fault-tolerance; we provided some examples and motivations in [11]. The novel part of the deontic logic presented here is that we introduce stratified levels of permissions, prohibition and obligation, which enables us to avoid having global normative constraints (i.e., the normative restrictions imposed in a component do not affect the other components in the system). These stratified levels of permission can also be used to express different levels of ideal behaviors; for the sake of simplicity we have not dealt with this in this paper.

We presented a simple example to illustrate the use of these ideas in practice. In this example, a variation of Dijkstra's philosophers, we use deontic predicates to state what the ideal behavior of philosophers are, the specification is built from in several components and they are then used (together with morphisms) to obtain the final design. Finally, it is worth mentioning that the logic is decidable and we have proposed a tableaux system for this logic in [14]; this will enable us to perform automatic analysis of specifications. We leave this as further work.

References

1. Fiadeiro, J.L., Maibaum, T.: Temporal theories as modularization units for concurrent system specification. Formal Aspects of Computing 4, 239–272 (1992)
2. Castro, P.F., Maibaum, T.: Deontic action logic, atomic boolean algebra and fault-tolerance. Journal of Applied Logic 7(4), 441–466 (2009)
3. Aqvist, L.: Deontic logic. In: Gabbay, D.M., Guenther, F. (eds.) Handbook of Philosophical Logic, vol. 2, pp. 605–714. Kluwer Academic Publishers, Dordrecht (1984)
4. Wieringa, R.J., Meyer, J.J.: Applications of deontic logic in computer science: A concise overview. Deontic Logic in Computer Science, Normative System Specification (1993)
5. Khosla, S., Maibaum, T.: The prescription and description of state-based systems. In: Banieqnal, B., Pnueli, H.A. (eds.) Temporal Logic in Computation. Springer, Heidelberg (1985)
6. Burstall, R., Goguen, J.: Putting theories together to make specifications. In: Reddy, R. (ed.) Porc. Fifth International Joint Conference on Artificial Intelligence (1977)
7. MacLane, S.: Categories for the Working Mathematician. Springer, Heidelberg (1998)
8. Fiadeiro, J.L., Maibaum, T.: Towards object calculi. In: Saake, G., Sernadas, A. (eds.) Information Systems; Correctness and Reusability, Technische Universität Braunschweig (1991)
9. Barringer, H.: The use of temporal logic in the compositional specification of concurrent systems. In: Galton, A. (ed.) Temporal Logic and their Applications. Academic Press, London (1987)
10. DeNicola, R., Vaandrager, F.: Three logics for branching bisimulation. Journal of the ACM 42, 458–487 (1995)
11. Castro, P.F.: Deontic Action Logics for the Specification and Analysis of Fault-Tolerance. PhD thesis, McMaster University, Department of Computing and Software (2009)
12. Goguen, J., Burstall, R.: Institutions: Abstract model theory for specification and programming. Journal of the Association of Computing Machinery (1992)
13. Fiadeiro, J.L., Sernadas, A.: Structuring theories on consequence. In: Recent Trends in Data Type Specification, 5th Workshop on Abstract Data Types, Gullane, Scotland, Selected Papers, pp. 44–72 (1987)
14. Castro, P.F., Maibaum, T.: A tableaux system for deontic action logic. In: van der Meyden, R., van der Torre, L. (eds.) DEON 2008. LNCS (LNAI), vol. 5076, pp. 34–48. Springer, Heidelberg (2008)

Justification Logic and History Based Computation*

Francisco Bavera[1] and Eduardo Bonelli[2]

[1] UNRC and CONICET, Argentina
fbavera@gmail.com
[2] UNQ and CONICET, Argentina
eabonelli@gmail.com

Abstract. Justification Logic (JL) is a refinement of modal logic that has recently been proposed for explaining well-known paradoxes arising in the formalization of Epistemic Logic. Assertions of knowledge and belief are accompanied by justifications: the formula $[\![t]\!]A$ states that t is "reason" for knowing/believing A. We study the computational interpretation of JL via the Curry-de Bruijn-Howard isomorphism in which the modality $[\![t]\!]A$ is interpreted as: t is a type derivation justifying the validity of A. The resulting lambda calculus is such that its terms are aware of the reduction sequence that gave rise to them. This serves as a basis for understanding systems, many of which belong to the security domain, in which computation is history-aware.

1 Introduction

This paper is concerned with the computational interpretation of *Justification Logic* [Art95, Art01, Art08] (**JL**). **JL** is a refinement of modal logic that has recently been proposed for explaining well-known paradoxes arising in the formalization of Epistemic Logic. Assertions of knowledge and belief are accompanied by *justifications*: the modality $[\![t]\!]A$ states that t is "reason" for knowing/believing A. The starting point of this work is the observation that if t is understood as a typing derivation of a term of type A, then a term of type $[\![t]\!]A$ should incorporate some encoding of t. Suppose this typing derivation is seen as a logical derivation in natural deduction. Then any normalisation steps applied to it would produce a new typing derivation for A and, moreover, its relation to t would have to be made explicit in order for derivations to be closed under normalisation (in type systems parlance: for Subject Reduction (SR) to hold). This suggests that the computational interpretation of **JL** is a lambda calculus, which we dub $\lambda^{\natural}$, that records its computation history. This work is an attempt at making these ideas precise.

We begin with some examples supplying informal explanations whenever appropriate (rigorous definitions must wait until the necessary background has been introduced). The expression $!_e^{\alpha_1,\dots,\alpha_n} M$ is called an *audited (computation) unit*, M being the *body*, e the *history* or *trail* of computation producing M and

* Work partially supported by ANPCyT PICT 2006-01216 and ITBA.

α_i, $i \in 1..n$, the *trail variables* that are used for consulting the history. Each reduction step that takes place in M updates e accordingly (except if this step is inside a nested audited unit). Consider the unit $!^{\alpha}_{\mathit{Rfl}(s)}(\lambda a : \mathbb{N}.a)\, b$. Its body consists of the identity function over the type $\mathbb{N}$ of the natural numbers applied to a variable b; $\mathit{Rfl}(s)$ is the empty trail with s being the encoding of a type derivation of $(\lambda a : \mathbb{N}.a)\, b$; the trail variable α plays no role in this example. This term reduces to $!^{\alpha}_{\mathit{Trn}(\beta(a^{\mathbb{N}}.a,b),\mathit{Rfl}(s))} b$. The new trail $\mathit{Trn}(\beta(a^{\mathbb{N}}.a, b), \mathit{Rfl}(s))$ indicates that a β step took place at the root; the Trn trail constructor indicates composition of trails.

Inspection of trails is achieved by means of trail variables. These variables are affine (i.e. at most one permitted use) since each trail lookup may produce a different result. Evaluation of trail variables inside an audited unit consists in first looking up the trail and then immediately traversing it replacing each constructor of the trail with a term of the appropriate type[1]. This mapping from trail constructors to terms is called a *trail replacement*. All occurrences of trail variables are thus written $\alpha\theta$ where α is a trail variable and θ a trail replacement. For example, suppose after a number of computation steps we attain the term $!^{\alpha}_{e}\mathsf{if}\ \alpha\theta > \underline{5}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ \underline{2}$, where e denotes the current history and $\underline{n}$ a numeral. Given the following definition of θ, $\alpha\theta$ counts the number of β steps that have taken place (expressions such as Tlk below are other trail constructors and may be ignored for the moment):

$$\begin{array}{rlrl}
\theta(\mathit{Rfl}) = \theta(\mathit{Tlk}) & \stackrel{\text{def}}{=} \underline{0} & \theta(\mathit{Rpl}) & \stackrel{\text{def}}{=} \lambda\overline{a} : \mathbb{N}.a_1 + \ldots + a_{10} \\
\theta(\mathit{Sym}) = \theta(\mathit{Abs}) & \stackrel{\text{def}}{=} \lambda a : \mathbb{N}.a & \theta(\beta_{\Box}) & \stackrel{\text{def}}{=} \underline{0} \\
\theta(\mathit{Trn}) = \theta(\mathit{App}) = \theta(\mathit{Let}) & \stackrel{\text{def}}{=} \lambda a : \mathbb{N}.\lambda b : \mathbb{N}.a + b & \theta(\beta) & \stackrel{\text{def}}{=} \underline{1}
\end{array}$$

Thus, the term decides either to compute $\underline{1}$ or $\underline{2}$ depending on whether the number of β steps that have taken place is greater than 5 or not. An interesting feature of $\lambda^{\natural}$ is how it manages persistence of trails. It is achieved by means of the $\mathsf{let}\, u = M\, \mathsf{in}\ N$ construct (M is the argument and N the body of the let) which eliminates audited units and operates as exemplified below. Let P be $\lambda a : \mathbb{N}.\mathsf{if}\ \alpha\theta > \underline{5}\ \mathsf{then}\ a\ \mathsf{else}\ \underline{2}$ and consider the following term where the expression $\langle u; \{\alpha/\gamma\}\rangle$ consists of an audited unit variable u and a trail variable renaming $\{\alpha/\gamma\}$:

$$\mathsf{let}\, u = !^{\alpha}_{e_1} P\ \mathsf{in}\ !^{\gamma}_{\mathit{Rfl}(t)}\langle u; \{\alpha/\gamma\}\rangle\, \underline{1}$$

In a $\beta_{\Box}$ reduction step, first u is replaced by the body P of the audited unit $!^{\alpha}_{e_1} P$ together with its history e_1, then all occurrences of trail variables α are replaced by γ, and finally e_1 is merged with the trail of the new host unit:

$$!^{\gamma}_{\mathit{Trn}(e_2,\mathit{Rfl}(t'))}(\lambda a : \mathbb{N}.\mathsf{if}\ \gamma\theta > \underline{5}\ \mathsf{then}\ a\ \mathsf{else}\ \underline{2})\, \underline{1}$$

Trail e_2 is $\mathit{Trn}(\beta_{\Box}(u.r_1, \alpha.r_2), \mathit{App}(e_1', \mathit{Rfl}(s)))$. Here $u.r_1$ and $\alpha.r_2$ are encodings of typing derivations for the body and argument of the let resp., e_1' is e_1 updated with $\{\alpha/\gamma\}$, s encodes a type derivation for $\underline{1}$ and t' is t where all occurrences of

[1] In the same way as one recurs over lists using fold in functional programming, replacing nil and cons by appropriate terms.

u have been replaced by the $\alpha.r_2$. The $\beta_\Box$ trail construct reflects a reduction of a let redex at the root. Note how, (1) the history of the unit $!^{\alpha}_{e_1} P$ has propagated to the new host unit (as the trail $App(e'_1, Rfl(s))$ which reflects that activity described by e'_1 has taken place in the left argument of the application $(\lambda a : \mathbb{N}.\mathsf{if}\ \gamma\theta > \underline{5}\ \mathsf{then}\ a\ \mathsf{else}\ \underline{2})\ \underline{1}$), and (2) how the trail variable α has been replaced by γ so that all subsequent trail lookups now correctly refer to the trail of the new host unit. All these operations arise from an analysis of the normalisation of **JL** derivations.

The contributions of this paper are a proof theoretical analysis of a λ-calculus which produces a trail of its execution. This builds on ideas stemming from work on **JL**, judgemental analysis of modal logic [ML96, DP96, DP01b, DP01a] and Contextual Modal Type Theory [NPP08]. More precisely, we argue how a fragment of **JL** whose notion of validity is relative to a context of affine variables of justification equivalence may be seen, via the Curry-de Bruijn-Howard interpretation, as a type system for a calculus that records its computation history.

Related work. S. Artemov introduced **JL** in [Art95, Art01, Art08]. For natural deduction and sequent calculus presentations consult [Art01, Bre01, AB07]. Computational interpretation of proofs in **JL** are studied in [AA01, AB07, BF09], however none of these address audit trails. From a type theoretic perspective we should mention the theory of dependent types [Bar92] where types may depend on terms, in much the same way that a type $[\![s]\!]A$ depends on the proof term s. However, dependent type theory lacks a notion of internalisation of derivations as is available in **JL**.

Structure of the paper. Sec. 2 introduces $\mathbf{JL}^{\bullet}$, an affine fragment of **JL**. Sec. 3 studies normalisation in this system. We then introduce a term assignment for this logic in order to obtain a lambda calculus with computation history trails. This calculus is endowed with a call-by-value operational semantics and type safety of this semantics w.r.t. the type system is proved. Sec. 5 addresses strong normalisation. Finally, we conclude and suggest further avenues for research.

2 The Logic

JL (formerly, the *Logic of Proofs*) is a modal logic of provability which has a sound and complete arithmetical semantics. This section introduces a natural deduction presentation for a fragment[2] of **JL**. The inference schemes we shall define give meaning to *hypothetical judgements with explicit evidence* $\Delta; \Gamma; \Sigma \vdash A \mid s$ whose intended reading is: "*s is evidence that A is true under validity hypothesis Δ, truth hypothesis Γ and equivalence hypothesis Σ*. Hypothesis in Γ are the standard term variables $a, b, \ldots$, those in Δ are audited unit variables $u, v, \ldots$, and those in Σ are trail variables $\alpha, \beta, \ldots$. These last hypothesis are often referred to as *equivalence hypothesis* since the type of a trail variable is a proposition that states the equivalence of two typing derivations. The syntax of each component of the judgement is as follows:

[2] Intuitionistic propositional **JL** without the "plus" polynomial proof constructor.

$$\frac{a : A \in \Gamma}{\Delta; \Gamma; \Sigma \vdash A \mid a}\,\mathsf{oVar} \qquad \frac{\Delta; \Gamma, a : A; \Sigma \vdash B \mid s}{\Delta; \Gamma; \Sigma \vdash A \supset B \mid \lambda a : A.s}\supset \mathsf{I}$$

$$\frac{\begin{array}{c}\Delta; \Gamma_1; \Sigma_1 \vdash A \supset B \mid s \\ \Delta; \Gamma_2; \Sigma_2 \vdash A \mid t\end{array}}{\Delta; \Gamma_{1,2}; \Sigma_{1,2} \vdash B \mid s \cdot t}\supset \mathsf{E} \qquad \frac{u : A[\Sigma] \in \Delta \quad \Sigma\sigma \subseteq \Sigma'}{\Delta; \Gamma; \Sigma' \vdash A \mid \langle u; \sigma\rangle}\,\mathsf{mVar}$$

$$\frac{\begin{array}{c}\Delta; \cdot; \Sigma \vdash A \mid s \\ \Delta; \cdot; \Sigma \vdash \mathsf{Eq}(A, s, t) \mid e\end{array}}{\Delta; \Gamma; \Sigma' \vdash [\![\Sigma.t]\!]A \mid \Sigma.t}\,\Box\mathsf{I} \qquad \frac{\begin{array}{c}\Delta; \Gamma_1; \Sigma_1 \vdash [\![\Sigma.r]\!]A \mid s \\ \Delta, u : A[\Sigma]; \Gamma_2; \Sigma_2 \vdash C \mid t\end{array}}{\Delta; \Gamma_{1,2}; \Sigma_{1,2} \vdash C^u_{\Sigma.r} \mid \textsc{let}\, u : A[\Sigma] = s \,\textsc{in}\, t}\,\Box\mathsf{E}$$

$$\frac{\alpha : \mathsf{Eq}(A) \in \Sigma \quad \Delta; \cdot; \cdot \vdash \mathcal{T}^B \mid \theta^w}{\Delta; \Gamma; \Sigma \vdash B \mid \alpha\theta^w}\,\mathsf{Tlk} \qquad \frac{\Delta; \Gamma; \Sigma \vdash A \mid s \quad \Delta; \Gamma; \Sigma \vdash \mathsf{Eq}(A, s, t) \mid e}{\Delta; \Gamma; \Sigma \vdash A \mid t}\,\mathsf{Eq}$$

Fig. 1. Explanation for Hypothetical Judgements with Explicit Evidence

$$\begin{array}{rlrl}
\text{Propositions } A & ::= P \mid A \supset A \mid [\![\Sigma.s]\!]A & \text{Renaming } \sigma & ::= \{\alpha_1/\beta_1, \ldots, \alpha_n/\beta_n\} \\
\text{Validity ctxt } \Delta & ::= \cdot \mid \Delta, u : A[\Sigma] & \text{Evidence } s & ::= a \mid \lambda a : A.s \mid s \cdot s \\
\text{Truth ctxt } \Gamma & ::= \cdot \mid \Gamma, a : A & & \mid\ \langle u; \sigma\rangle \mid \Sigma.s \\
\text{Equiv. ctxt } \Sigma & ::= \cdot \mid \Sigma, \alpha : \mathsf{Eq}(A) & & \mid\ \textsc{let}\, u : A[\Sigma] = s \,\textsc{in}\, s \mid \alpha\theta^w
\end{array}$$

Both Γ and Σ are affine hypothesis whereas those in Δ are intuitionistic. Contexts are considered multisets; "$\cdot$" denotes the empty context. In $\Delta; \Gamma; \Sigma$ we assume all variables to be fresh. Variables in Σ are assigned a type of the form $\mathsf{Eq}(A)$[3]. A proposition is either a propositional variable P, an implication $A \supset B$ or a modality $[\![\Sigma.s]\!]A$. In $[\![\Sigma.s]\!]A$, "$\Sigma.$" binds all occurrences of trail variables in s and hence may be renamed at will. We refer to an encoding of a type derivation as *evidence*. Evidence bear witness to proofs of propositions, they encode each possible scheme that may be applied: truth hypothesis, abstraction, audited computation unit variable, audited computation unit introduction and elimination, and trail look-up. The expression θ^w ('w' is for 'witness') will be explained shortly. We write σ for trail variable (bijective) renamings. Free truth variables of s ($\mathsf{fvT}(s)$), free validity variables of s ($\mathsf{fvV}(s)$) and free trail variables of s ($\mathsf{fvTrl}(s)$) are defined as expected. We write s^a_t for the substitution of all free occurrences of a in s by t. Substitution of validity variables is denoted $s^u_{\Sigma.t}$. Its definition is standard except perhaps for the case $\langle u; \sigma\rangle^u_{\Sigma.s}$: here all occurrences of Σ in s are renamed via σ.

The meaning of hypothetical judgements with explicit evidence is given in Fig. 1 and determine the $\mathbf{JL}^\bullet$ *system*. The axiom scheme oVar states that judgement "$\Delta; \Gamma; \Sigma \vdash A \mid a$" is evident in itself: if we assume a is evidence that

[3] The type $\mathsf{Eq}(A)$ in an assignment $\alpha : \mathsf{Eq}(A)$ may informally be understood as $\exists x, y.\mathsf{Eq}(A, x, y)$ (where x, y stand for arbitrary type derivations of propositions of type A) since α stands for a proof of equivalence of two type derivations of propositions of type A about which nothing more may be assumed. These type derivations are hidden since trail lookup may take place at any time.

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi_1}{\Delta; a : A \vdash B \mid s}}{\Delta; \cdot \vdash A \supset B \mid \lambda a : A.s} \supset \mathsf{I} \qquad \dfrac{\pi_2}{\Delta; \cdot \vdash A \mid t}}{\Delta; \cdot \vdash B \mid (\lambda a : A.s) \cdot t} \supset \mathsf{E}}{\Delta; \Gamma \vdash [\![(\lambda a : A.s) \cdot t]\!]B \mid !(\lambda a : A.s) \cdot t} \Box\mathsf{I} \qquad \dfrac{\dfrac{\pi_3}{\Delta; \cdot \vdash B \mid s^a_t}}{\Delta; \Gamma \vdash \begin{array}{c}[\![(\lambda a : A.s) \cdot t]\!]B \\ !(\lambda a : A.s) \cdot t\end{array} \mid} \Box\mathsf{I}$$

Fig. 2. Failure of subject reduction for naive modal introduction scheme

proposition A is true, then we may immediately conclude that A is true with evidence a. The introduction scheme for the modality internalises meta-level evidence into the object logic. It arises from addressing the shortcomings of the more naive scheme (in which Σ is ignored) for introducing this modality:

$$\dfrac{\Delta; \cdot \vdash A \mid s}{\Delta; \Gamma \vdash [\![s]\!]A \mid !s} \Box\mathsf{I}$$

The resulting system is not closed under substitution of derivations. Eg. contraction of the derivation in Fig. 2 (left) would produce the invalid (since evidence s^a_t and $(\lambda a : A.s) \cdot t$ do not coincide) derivation of Fig. 2 (right) where π_3 is obtained from $\pi_{1,2}$ and an appropriate substitution principle. Subject Reduction may be regained, however, by introducing a judgement stating equivalence of evidence $\Delta; \Gamma; \Sigma \vdash \mathsf{Eq}(A, s, t) \mid e$, e is dubbed *equivalence witness*, together with the scheme Eq (Fig. 1). A consequence of this is that normalisation gives rise to instances of Eq appearing in any part of the derivation complicating metatheoretic reasoning. However, since this scheme can be permuted past all other schemes *except* for the introduction of the modality (as may easily be verified), this suggests postulating a hypothesis of evidence equivalence in the introduction scheme for the modality and results in the current scheme $\Box\mathsf{I}$ (Fig. 1). Normalisation steps performed in the derivation ending in the leftmost hypothesis are encoded by equivalence witness e. Finally, $\Box\mathsf{E}$ allows the discharging of validity hypothesis: to discharge the validity hypothesis $v : A[\Sigma]$, a proof of the validity of A under derivation equivalence assumptions Σ is required. In our system, this requires proving that $[\![\Sigma.r]\!]A$ is true with some evidence s.

A sample of the schemes defining evidence equivalence are given in Fig. 3. There are four evidence equivalence axioms (EqRefl, $\mathsf{Eq}\beta$, $\mathsf{Eq}\beta_\Box$ and EqTlk; the third is not exhibited) and six inference schemes (the rest). The axioms are used for recording principle contractions (Sec. 3) at the root of a term and schemes EqAbs, EqApp, EqLet and EqRpl (only second exhibited) enable the same recording but under each of the term constructors. In accordance with the abovementioned discussion on permutation of Eq past other schemes, there is no congruence scheme for the modality. Equivalence witness may be one of the following where $Rpl(e_1, \ldots, e_{10})$ is usually abbreviated $Rpl(\overline{e})$:

$$\begin{array}{rl} e ::= & Rfl(s) \mid Sym(e) \mid Trn(e, e) \mid \beta(a^A.s, s) \mid \beta_\Box(u^{A[\Sigma]}.s, \Sigma.s) \\ \mid & Trl(\theta^w, \alpha) \mid Abs(a^A.e) \mid App(e, e) \mid Let(u^{A[\Sigma]}.e, e) \mid Rpl(e_1, \ldots, e_{10}) \end{array}$$

$$\frac{\Delta;\Gamma;\Sigma \vdash A \mid s}{\Delta;\Gamma;\Sigma \vdash \mathsf{Eq}(A,s,s) \mid \mathit{Rfl}(s)}\ \mathsf{EqRefl} \qquad \frac{\begin{array}{c}\Delta;\Gamma;\Sigma \vdash \mathsf{Eq}(A,s_1,s_2) \mid e_1 \\ \Delta;\Gamma;\Sigma \vdash \mathsf{Eq}(A,s_2,s_3) \mid e_2\end{array}}{\Delta;\Gamma;\Sigma \vdash \mathsf{Eq}(A,s_1,s_3) \mid \mathit{Trn}(e_1,e_2)}\ \mathsf{EqTrans}$$

$$\frac{\begin{array}{c}\Delta;\Gamma_1,a:A;\Sigma_1 \vdash B \mid s \\ \Delta;\Gamma_2;\Sigma_2 \vdash A \mid t\end{array}}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash \mathsf{Eq}(B,s^a_t,(\lambda a:A.s)\cdot t) \mid \beta(a^A.s,t)}\ \mathsf{Eq}\beta$$

$$\frac{\Delta;\cdot;\Sigma_1 \vdash \mathsf{Eq}(A,s,t) \mid e \quad \Delta;\cdot;\cdot \vdash \mathcal{T}^B \mid \theta^w \quad \alpha:\mathsf{Eq}(A) \in \Sigma_2}{\Delta;\Gamma;\Sigma_2 \vdash \mathsf{Eq}(B,e\theta^w,\alpha\theta^w) \mid \mathit{Trl}(\theta^w,\alpha)}\ \mathsf{EqTlk}$$

$$\frac{\begin{array}{c}\Delta;\Gamma_1;\Sigma_1 \vdash \mathsf{Eq}(A\supset B,s_1,s_2) \mid e_1 \\ \Delta;\Gamma_2;\Sigma_2 \vdash \mathsf{Eq}(A,t_1,t_2) \mid e_2\end{array}}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash \mathsf{Eq}(B,s_1\cdot t_1,s_2\cdot t_2) \mid \mathit{App}(e_1,e_2)}\ \mathsf{EqApp}$$

Fig. 3. Sample schemes defining evidence equivalence judgement

Regarding trail look-up (Tlk in Fig. 1) recall from the introduction that we append each reference to a trail variable with a trail replacement. Therefore, the evidence for look-ups has to be accompanied by proofs of propositions corresponding to each term that is to replace equivalence witness constructors. The evidence for each of these proofs is then encoded as θ^w. This requirement is reflected by the hypothesis $\Delta;\cdot;\cdot \vdash \mathcal{T}^B \mid \theta^w$ which is a shorthand for $\Delta;\cdot;\cdot \vdash \mathcal{T}^B(c) \mid \theta^w(c)$, for each c in the set of equivalence witness constructors $\{\mathit{Rfl}, \mathit{Sym}, \mathit{Trn}, \beta, \beta_\Box, \mathit{Tlk}, \mathit{Abs}, \mathit{App}, \mathit{Let}, \mathit{Rpl}\}$, where $\mathcal{T}^B(c)$ is the type of term that replaces the trail constructor c. These types are defined as one might expect (for example, $\mathcal{T}^B(\mathit{Trn}) \overset{\text{def}}{=} B \supset B \supset B$ and $\mathcal{T}^B(\beta) \overset{\text{def}}{=} B$).

Some basic meta-theoretic results about $\mathbf{JL}^\bullet$ are presented next. The judgements in the statement of these results are decorated with terms (eg. M) which may safely be ignored for the time being (they are introduced in Sec. 4).

Lemma 1 (Weakening)

1. *If $\Delta;\Gamma;\Sigma \vdash M : A \mid s$ is derivable, then so is $\Delta';\Gamma';\Sigma' \vdash M : A \mid s$, where $\Delta \subseteq \Delta'$, $\Gamma \subseteq \Gamma'$ and $\Sigma \subseteq \Sigma'$.*
2. *If $\Delta;\Gamma;\Sigma \vdash \mathsf{Eq}(A,s,t) \mid e$ is derivable, then so is $\Delta';\Gamma';\Sigma' \vdash \mathsf{Eq}(A,s,t) \mid e$, where $\Delta \subseteq \Delta'$, $\Gamma \subseteq \Gamma'$ and $\Sigma \subseteq \Sigma'$.*

We abbreviate Γ_1, Γ_2 with $\Gamma_{1,2}$. If $\Gamma = \Gamma_1, a : A, \Gamma_3$, we write $\Gamma^a_{\Gamma_2}$ for $\Gamma_{1,2,3}$.

Lemma 2 (Subst. Principle for Truth Hypothesis). *Suppose $\Delta;\Gamma_2;\Sigma_2 \vdash N : A \mid t$ is derivable and $a : A \in \Gamma_1$.*

1. *If $\Delta;\Gamma_1;\Sigma_1 \vdash M : B \mid s$, then $\Delta;{\Gamma_1}^a_{\Gamma_2};\Sigma_{1,2} \vdash M^a_{N,t} : B \mid s^a_t$.*
2. *If $\Delta;\Gamma_1;\Sigma_1 \vdash \mathsf{Eq}(B,s_1,s_2) \mid e$, then $\Delta;{\Gamma_1}^a_{\Gamma_2};\Sigma_{1,2} \vdash \mathsf{Eq}(B,(s_1)^a_t,(s_2)^a_t) \mid e^a_t$.*

In the substitution principle for validity variables, note that substitution of $u : A[\Sigma_1]$ requires not only a derivation of $\Delta_{1,2}; \cdot; \Sigma_1 \vdash M : A \mid s$, but also its normalisation history $\Delta_{1,2}; \cdot; \Sigma_1 \vdash \mathsf{Eq}(A, s, t) \mid e_1$ (cf. substitution of validity variables, in particular the clause for $\langle u; \sigma \rangle$, in Sec. 4).

Lemma 3 (Subst. Principle for Validity Hypothesis). *Suppose judgements* $\Delta_{1,2}; \cdot; \Sigma_1 \vdash M : A \mid s$ *and* $\Delta_{1,2}; \cdot; \Sigma_1 \vdash \mathsf{Eq}(A, s, t) \mid e_1$ *are derivable. Let* $\Delta \overset{\text{def}}{=} \Delta_1, u : A[\Sigma_1], \Delta_2$*. Then:*

1. *If* $\Delta; \Gamma; \Sigma_2 \vdash N : C \mid r$*, then* $\Delta_{1,2}; \Gamma; \Sigma_2 \vdash N^u_{\Sigma_1.(M,t,e_1)} : C^u_{\Sigma_1.t} \mid r^u_{\Sigma_1.t}$.
2. *If* $\Delta; \Gamma; \Sigma_2 \vdash \mathsf{Eq}(C, s_1, s_2) \mid e_2$*, then* $\Delta_{1,2}; \Gamma; \Sigma_2 \vdash \mathsf{Eq}(C^u_{\Sigma_1.t}, {s_1}^u_{\Sigma_1.t}, {s_2}^u_{\Sigma_1.t}) \mid {e_2}^u_{\Sigma_1.t}$.

The last ingredient we require before discussing normalisation is the following lemma which is used for computing the results of trail look-up. The expression $e\theta$ produces a term by replacing each equivalence witness constructor c in e by its correesponding term $\theta(c)$. For example, $\beta(a^A.r, t)\theta \overset{\text{def}}{=} \theta(\beta)$ and $Trn(e_1, e_2)\theta \overset{\text{def}}{=} \theta(Trn)\, e_1\theta\, e_2\theta$. In contrast, $e\theta^w$ produces evidence by replacing each equivalence witness constructor c in e with $\theta^w(c)$.

Lemma 4. $\Delta; \cdot; \cdot \vdash \mathcal{T}^B \mid \theta^w$ *and* $\Delta; \cdot; \Sigma_2 \vdash \mathsf{Eq}(A, s, t) \mid e$ *implies* $\Delta; \cdot; \cdot \vdash e\theta : B \mid e\theta^w$*.*

3 Normalisation

Normalisation equates derivations and since $\mathbf{JL}^\bullet$ internalises its own proofs, normalisation steps must explicitly relate evidence in order for SR to hold. Normalisation is modeled as a two step process. First a *principle contraction* is applied, then a series of *permutation conversions* follow. Principle contractions introduce explicit witnesses of derivation equivalence. Permutation conversions standardize derivations by moving these witnesses to the innermost $\Box$ introduction scheme. There are three principal contractions (β, $\beta_\Box$ and Tlk-contraction), the first two of which rely on the substitution principles discussed earlier. The first replaces a derivation of the form:

$$\dfrac{\dfrac{\dfrac{\pi_1}{\Delta; \Gamma_1, a : A; \Sigma_1 \vdash B \mid s}}{\Delta; \Gamma_1; \Sigma_1 \vdash A \supset B \mid \lambda a : A.s} \supset \mathsf{I} \qquad \dfrac{\pi_2}{\Delta; \Gamma_2; \Sigma_2 \vdash A \mid t}}{\Delta; \Gamma_{1,2}; \Sigma_{1,2} \vdash B \mid (\lambda a : A.s) \cdot t} \supset \mathsf{E}$$

by the following, where π_3 is a derivation of $\Delta; \Gamma_{1,2}; \Sigma_{1,2} \vdash B \mid s^a_t$ resulting from π_1 and π_2 and the Substitution Principle for Truth Hypothesis:

$$\dfrac{\pi_3 \qquad \dfrac{\dfrac{\pi_1}{\Delta; \Gamma_1, a : A; \Sigma_1 \vdash B \mid s} \qquad \dfrac{\pi_2}{\Delta; \Gamma_2; \Sigma_2 \vdash A \mid t}}{\Delta; \Gamma_{1,2}; \Sigma_{1,2} \vdash \mathsf{Eq}(B, s^a_t, (\lambda a : A.s) \cdot t) \mid \beta(a^A.s, t)}}{\Delta; \Gamma_{1,2}; \Sigma_{1,2} \vdash B \mid (\lambda a : A.s) \cdot t} \mathsf{Eq}$$

The second contraction replaces:

$$\dfrac{\dfrac{\begin{array}{c}\Delta;\cdot;\Sigma \vdash A \mid s\\ \Delta;\cdot;\Sigma \vdash \mathsf{Eq}(A,s,t) \mid e_1\end{array}}{\Delta;\Gamma_1;\Sigma_1 \vdash [\![\Sigma.t]\!]A \mid \Sigma.t}\ \Box\mathsf{I} \qquad \Delta,u:A[\Sigma];\Gamma_2;\Sigma_2 \vdash C \mid r}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash C^u_{\Sigma.t} \mid \text{LET}\, u : A[\Sigma] = \Sigma.t\, \text{IN}\, r}\ \Box\mathsf{E}$$

with the following derivation where π is a derivation of $\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash C^u_{\Sigma.t} \mid t^u_{\Sigma.t}$ resulting from the Substitution Principle for Validity Hypothesis followed by weakening (of Γ_1 and Σ_1) and e_2 is $\beta_\Box(u^{A[\Sigma_1]}.r, \Sigma.t)$:

$$\dfrac{\pi \qquad \dfrac{\begin{array}{c}\Delta;\cdot;\Sigma \vdash A \mid s\\ \Delta;\cdot;\Sigma \vdash \mathsf{Eq}(A,s,t) \mid e_1\\ \Delta,u:A[\Sigma];\Gamma_2;\Sigma_2 \vdash C \mid r\end{array}}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash \mathsf{Eq}(C^u_{\Sigma.t}, r^u_{\Sigma.t}, \text{LET}\, u : A[\Sigma] = \Sigma.t\, \text{IN}\, r) \mid e_2}\ \mathsf{Eq}\beta_\Box}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash C^u_{\Sigma.t} \mid \text{LET}\, u : A[\Sigma] = \Sigma.t\, \text{IN}\, r}\ \mathsf{Eq}$$

Tlk-contraction models audit trail look-up. Consider the following derivation, where $\Sigma_1 \subseteq \Sigma_2$, $\Delta' \subseteq \Delta$ and the branch from the depicted instance of Tlk in π_1 to its conclusion has no instances of $\Box\mathsf{I}$:

$$\dfrac{\begin{array}{c}\dfrac{\begin{array}{c}\alpha : \mathsf{Eq}(A) \in \Sigma_1\\ \Delta;\cdot;\cdot \vdash \mathcal{T}^B \mid \theta^w\end{array}}{\Delta;\Gamma;\Sigma_1 \vdash B \mid \alpha\theta^w}\ \mathsf{Tlk}\\ \vdots\ \pi_1\\ \Delta';\cdot;\Sigma_2 \vdash A \mid s\end{array} \qquad \dfrac{\pi_2}{\Delta';\cdot;\Sigma_2 \vdash \mathsf{Eq}(A,s,t) \mid e}}{\Delta';\Gamma';\Sigma' \vdash [\![\Sigma_2.t]\!]A \mid \Sigma_2.t}\ \Box\mathsf{I}$$

The instance of Tlk in π_1 is replaced by the following derivation where π_2' is obtained from π_2 by resorting to Lem. 4 and Lem. 1. Also, $\Delta;\cdot;\Sigma_2 \vdash \mathsf{Eq}(A,s,t) \mid e$ is obtained from $\Delta';\cdot;\Sigma_2 \vdash \mathsf{Eq}(A,s,t) \mid e$ by Lem. 1.

$$\dfrac{\dfrac{\pi_2'}{\Delta;\Gamma;\Sigma_1 \vdash B \mid e\theta^w} \qquad \dfrac{\begin{array}{c}\Delta;\cdot;\Sigma_2 \vdash \mathsf{Eq}(A,s,t) \mid e\\ \Delta;\cdot;\cdot \vdash \mathcal{T}^B \mid \theta^w\end{array}}{\Delta;\Gamma;\Sigma_1 \vdash \mathsf{Eq}(B, e\theta^w, \alpha\theta^w) \mid \mathit{Trl}(\theta^w,\alpha)}\ \mathsf{EqTlk}}{\Delta;\Gamma;\Sigma_1 \vdash B \mid \alpha\theta^w}\ \mathsf{Eq}$$

As for the permutation conversions, they indicate how Eq is permuted past any of the inference schemes in $\{\supset\mathsf{I}, \supset\mathsf{E}, \Box\mathsf{E}, \mathsf{Eq}, \mathsf{Tlk}\}$. Also, there is a conversion that fuses Eq just above the left hypothesis in an instance of $\Box\mathsf{I}$ with the trail of the corresponding unit is also coined. As an example Eq permutes past $\supset\mathsf{I}$ by replacing:

$$\dfrac{\dfrac{\dfrac{\pi_1}{\Delta;\Gamma,a:A;\Sigma \vdash B \mid s} \qquad \dfrac{\pi_2}{\Delta;\Gamma,a:A;\Sigma \vdash \mathsf{Eq}(B,s,t) \mid e}}{\Delta;\Gamma,a:A;\Sigma \vdash B \mid t}\ \mathsf{Eq}}{\Delta;\Gamma;\Sigma \vdash A \supset B \mid \lambda a : A.t}\ \supset\mathsf{I}$$

with the following derivation where π_3 is a derivation of $\Delta;\Gamma;\Sigma \vdash A \supset B \mid \lambda a : A.s$ obtained from π_1 and $\supset \mathsf{I}$:

$$\dfrac{\pi_3 \qquad \dfrac{\Delta;\Gamma, a : A;\Sigma \vdash \mathsf{Eq}(B,s,t) \mid e}{\Delta;\Gamma;\Sigma \vdash \mathsf{Eq}(A \supset B, \lambda a : A.s, \lambda a : A.t) \mid Abs(a^A.e)}\,\mathsf{EqAbs}}{\Delta;\Gamma;\Sigma \vdash A \supset B \mid \lambda a : A.t}\,\mathsf{Eq}$$

4 Term Assignment

Computation by normalisation is non-confluent, as one might expect (audit trail look-up affects computation), hence a strategy is required. This section introduces the call-by-value $\lambda^{\natural}$-calculus. It is obtained via a term assignment for $\mathbf{JL}^{\bullet}$. The syntax of $\lambda^{\natural}$ terms is:

$$M ::= a \mid \lambda a : A.M \mid M\,M \mid \langle u;\sigma\rangle \mid !^{\Sigma}_{e} M \mid \mathsf{let}\, u : A[\Sigma] = M \,\mathsf{in}\, M \mid \alpha\theta \mid e \rhd M$$

In addition to term variables, abstraction and application we also have audited computation unit variables, audited computation units, audited computation unit substitution, trail look-up and terms decorated with equivalence witnesses. We occasionally drop the type decoration in let construct for readability. Since terms may be decorated with equivalence witnesses, substitution (both for truth and validity hypothesis) substitutes free occurrences of variables with both terms and evidence. We write $M^{a}_{N,t}$ for substitution of truth variables and $M^{u}_{\Sigma.(N,t,e)}$ for substitution of validity variables (similar notions apply to substitution in propositions, evidence and equivalence witnesses). Note that "$\Sigma.$" in $\Sigma.(N,t,e)$ binds all free occurrences of trail variables from Σ which occur in N, t and e. For illustration we give the definition of $M^{u}_{\Sigma.(N,t,e)}$, where $s^{u}_{\Sigma.t}$ traverses the structure of s replacing $\langle u;\sigma\rangle^{u}_{\Sigma.s}$ with $s\sigma$ and $e^{u}_{\Sigma.t}$ traverses the structure of e until it reaches one of $Rfl(r_1)$, $\beta(a^A.r_1, r_2)$ or $\beta_{\Box}(v^{A[\Sigma']}.r_1, \Sigma'.r_2)$ in which case it resorts to substitution over the r_is. Note how the fourth clause of the definition of $M^{u}_{\Sigma.(N,t,e)}$ below substitutes $\langle u;\sigma\rangle$ with $e\sigma \rhd N\sigma$, thus propagating the history.

$$\begin{array}{rcl@{\qquad}rcl}
b^{u}_{\Sigma.(N,t,e)} & \stackrel{\text{def}}{=} & b & (!^{\Sigma'}_{e'} M)^{u}_{\Sigma.(N,t,e)} & \stackrel{\text{def}}{=} & !^{\Sigma'}_{e'^{u}_{\Sigma.t}} M^{u}_{\Sigma.(N,t,e)} \\
(\lambda b : A.M)^{u}_{\Sigma.(N,t,e)} & \stackrel{\text{def}}{=} & \lambda b : A.M^{u}_{\Sigma.(N,t,e)} & (\text{LET}\, v = P & \stackrel{\text{def}}{=} & \text{LET}\, v = P^{u}_{\Sigma.(N,t,e)} \\
(P\,Q)^{u}_{\Sigma.(N,t,e)} & \stackrel{\text{def}}{=} & P^{u}_{\Sigma.(N,t,e)}\, Q^{u}_{\Sigma.(N,t,e)} & \text{IN}\, Q)^{u}_{\Sigma.(N,t,e)} & & \text{IN}\, Q^{u}_{\Sigma.(N,t,e)} \\
\langle u;\sigma\rangle^{u}_{\Sigma.(N,t,e)} & \stackrel{\text{def}}{=} & e\sigma \rhd N\sigma & (\alpha\theta)^{u}_{\Sigma.(N,t,e)} & \stackrel{\text{def}}{=} & \alpha(\theta^{u}_{\Sigma.(N,t,e)}) \\
\langle v;\sigma\rangle^{u}_{\Sigma.(N,t,e)} & \stackrel{\text{def}}{=} & \langle v;\sigma\rangle & (e' \rhd M)^{u}_{\Sigma.(N,t,e)} & \stackrel{\text{def}}{=} & e'^{u}_{\Sigma.t} \rhd M^{u}_{\Sigma.(N,t,e)}
\end{array}$$

The typing judgement $\Delta;\Gamma;\Sigma \vdash M : A \mid s$ is defined by means of the *typing schemes* obtained from decorating the inference schemes of Fig. 1 with terms. Sample schemes are given in Fig. 4. A term M is said to be *typable* if there exists $\Delta, \Gamma, \Sigma, A, s$ s.t. $\Delta;\Gamma;\Sigma \vdash M : A \mid s$ is derivable. The operational semantics of $\lambda^{\natural}$ is specified by a binary relation over typed terms called *reduction* ($M \to N$). In order to define reduction we first introduce two intermediate notions, namely *principle reduction* ($M \mapsto N$) and *permutation reduction* ($M \curvearrowright N$) . The former corresponds to principle contraction and the latter to permutation conversions of the normalisation procedure. The set of *evaluation contexts* and *values* are:

$$\frac{\Delta;\cdot;\Sigma \vdash M : A \mid s \quad \Delta;\cdot;\Sigma \vdash \mathsf{Eq}(A,s,t) \mid e}{\Delta;\Gamma;\Sigma' \vdash !^{\Sigma}_{e} M : [\![\Sigma.t]\!]A \mid \Sigma.t}\,\mathsf{TBox} \qquad \frac{\alpha : \mathsf{Eq}(A) \in \Sigma \quad \Delta;\cdot;\cdot \vdash \theta : \mathcal{T}^{B} \mid \theta^{w}}{\Delta;\Gamma;\Sigma \vdash \alpha\theta : B \mid \alpha\theta^{w}}\,\mathsf{TTlk}$$

$$\frac{\Delta;\Gamma_1;\Sigma_1 \vdash M : [\![\Sigma.r]\!]A \mid s \quad \Delta, u : A[\Sigma];\Gamma_2;\Sigma_2 \vdash N : C \mid t}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash \begin{array}{c}\mathsf{let}\, u : A[\Sigma] = M \,\mathsf{in}\, N : C^{u}_{\Sigma.r} \\ \mathrm{LET}\, u : A[\Sigma] = s \,\mathrm{IN}\, t\end{array} \mid}\,\mathsf{TLetB} \qquad \frac{\Delta;\Gamma;\Sigma \vdash M : A \mid s \quad \Delta;\Gamma;\Sigma \vdash \mathsf{Eq}(A,s,t) \mid e}{\Delta;\Gamma;\Sigma \vdash e \rhd M : A \mid t}\,\mathsf{TEq}$$

Fig. 4. Sample typing schemes for $\lambda^{\natural}$

$$\begin{array}{rcl} \mathcal{E} & ::= & \Box \mid \mathcal{E}\ M \mid V\ \mathcal{E} \mid \mathsf{let}\, u : A[\Sigma] = \mathcal{E} \,\mathsf{in}\, M \\ & \mid & !^{\Sigma}_{e}\mathcal{E} \mid \alpha\{c_1/V_1, \ldots, c_j/V_j, c_{j+1}/\mathcal{E}, \ldots\} \\ \mathcal{F} & ::= & \Box \mid \mathcal{F}\ M \mid V\ \mathcal{F} \mid \mathsf{let}\, u : A[\Sigma] = \mathcal{F} \,\mathsf{in}\, M \end{array} \qquad \begin{array}{rcl} V & ::= & a \mid \langle u;\sigma\rangle \mid \lambda a : A.M \\ & \mid & !^{\Sigma}_{e} V \\ \theta_V & ::= & \{c_1/V_1, \ldots, c_{10}/V_{10}\} \end{array}$$

Evaluation contexts are represented with letters $\mathcal{E}, \mathcal{E}'$, etc. Note that reduction under the audited unit constructor is allowed. Contexts $\mathcal{F}$ are required for defining $\mathcal{L}$, the principle reduction axiom for trail look-up (defined below). It differs from $\mathcal{E}$ by not allowing holes under the audited unit constructor. The set of values are standard except for $!^{\Sigma}_{e}V$: audited units with fully evaluated bodies are also values. θ_V is a trail replacement consisting entirely of values. Principle reduction is presented by means of the following principle reduction *axiom* and *congruence schemes*:

$$\begin{array}{rcl} (\lambda a : A.M)\ V & \rightharpoonup_{\beta} & \beta(a^{A}.s, t) \rhd M^{a}_{V,t} \\ \mathsf{let}\, u : A[\Sigma] = !^{\Sigma}_{e} V \,\mathsf{in}\, N & \rightharpoonup_{\beta_\Box} & \beta_\Box(u^{A[\Sigma]}.t, \Sigma.s) \rhd N^{u}_{\Sigma.(V,s,e)} \\ !^{\Sigma}_{e}\mathcal{F}[\alpha\theta_V] & \rightharpoonup_{\mathcal{L}} & !^{\Sigma}_{e}\mathcal{F}[\mathit{Trl}(\theta^{w}_{V}, \alpha) \rhd e\theta_V] \end{array}$$

$$M \rightharpoonup N \text{ implies } \mathcal{E}[M] \mapsto \mathcal{E}[N]7$$

These schemes have been abridged by removing typing information. For example, the fully decorated presentation of β is:

$$\frac{\Delta;\Gamma_1, a : A;\Sigma_1 \vdash M : B \mid s \quad \Delta;\Gamma_2;\Sigma_2 \vdash V : A \mid t}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash (\lambda a : A.M)\ V \rightharpoonup_{\beta} \beta(a^{A}.s, t) \rhd M^{a}_{V,t} : B \mid (\lambda a : A.s) \cdot t}$$

The fully decorated presentation of $\beta_\Box$ is as follows where $O \overset{\text{def}}{=} \mathsf{let}\, u : A[\Sigma] = !^{\Sigma}_{e} V \,\mathsf{in}\, N$ and $P \overset{\text{def}}{=} \beta_\Box(u^{A[\Sigma]}.t, \Sigma.s) \rhd N^{u}_{\Sigma.(V,s,e)}$:

$$\frac{\Delta;\cdot;\Sigma \vdash V : A \mid r \quad \Delta;\cdot;\Sigma \vdash \mathsf{Eq}(A,r,s) \mid e_1 \quad \Delta, u : A[\Sigma];\Gamma_2;\Sigma_2 \vdash N : C \mid t}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash O \rightharpoonup_{\beta_\Box} P : C^{u}_{\Sigma.s} \mid \mathrm{LET}\, u : A[\Sigma] = \Sigma.s \,\mathrm{IN}\, t}$$

Each principle reduction scheme produces a trail of its execution. Note that $\beta_\Box$ replaces all occurrences of $\langle u;\sigma\rangle$ with $e\sigma \rhd V\sigma$, correctly: (1) preserving trails and (2) rewiring trail variables so that they now refer to their host audited computation unit. Regarding permutation reduction, the original schemes obtained from the normalisation procedure are the contextual closure of the first group

$$\begin{array}{rcl}
(e \rhd M)\ N & \curvearrowright & App(e, Rfl(t)) \rhd (M\ N) \\
M\ (e \rhd N) & \curvearrowright & App(Rfl(t), e) \rhd (M\ N) \\
\lambda a : A.(e \rhd M) & \curvearrowright & Abs(a.e) \rhd (\lambda a : A.M) \\
\mathsf{let}\, u = (e \rhd M)\, \mathsf{in}\ N & \curvearrowright & Let(u.e, Rfl(t)) \rhd (\mathsf{let}\, u = M\, \mathsf{in}\ N) \\
\mathsf{let}\, u = M\, \mathsf{in}\ (e \rhd N) & \curvearrowright & Let(u.Rfl(s), e) \rhd (\mathsf{let}\, u = M\, \mathsf{in}\ N) \\
!^{\Sigma}_{e_2}(e_1 \rhd M) & \curvearrowright & !^{\Sigma}_{Trn(e_1,e_2)} M \\
e_1 \rhd (e_2 \rhd M) & \curvearrowright & Trn(e_1, e_2) \rhd M
\end{array}$$

$$\begin{array}{rcl}
Trn(App(e_1, e_2), App(e_3, e_4)) & \curvearrowright & App(Trn(e_1, e_3), Trn(e_2, e_4)) \\
Trn(Abs(a.e_1), Abs(a.e_2)) & \curvearrowright & Abs(a.Trn(e_1, e_2)) \\
Trn(Let(u.e_1, e_2), Let(u.e_3, e_4)) & \curvearrowright & Let(u.Trn(e_1, e_3), Trn(e_2, e_4)) \\
Trn(Rfl(s), e) & \curvearrowright & e \\
Trn(e, Rfl(t)) & \curvearrowright & e \\
Trn(Trn(e_1, e_2), e_3) & \curvearrowright & Trn(e_1, Trn(e_2, e_3)) \\
Trn(App(e_1, e_2), Trn(App(e_3, e_4), e_5)) & \curvearrowright & Trn(App(Trn(e_1, e_3), Trn(e_2, e_4)), e_5) \\
Trn(Abs(a.e_1), Trn(Abs(a.e_2), e_3)) & \curvearrowright & Trn(Abs(a.Trn(e_1, e_2)), e_3) \\
Trn(Let(u.e_1, e_2), Trn(Let(u.e_3, e_4), e_5)) & \curvearrowright & Trn(Let(u.Trn(e_1, e_3), Trn(e_2, e_4)), e_5)
\end{array}$$

Fig. 5. Permutation reduction schemes

of rules depicted in Fig. 5[4]. As in principle reduction, these schemes operate on typed terms and have been abridged. Eg. the full presentation of the first is:

$$\frac{\Delta; \Gamma_1; \Sigma_1 \vdash M : A \supset B \mid r \quad \Delta; \Gamma_1; \Sigma_1 \vdash \mathsf{Eq}(A \supset B, r, s) \mid e \quad \Delta; \Gamma_2; \Sigma_2 \vdash N : A \mid t}{\Delta; \Gamma_{1,2}; \Sigma_{1,2} \vdash (e \rhd M)\, N \curvearrowright App(e, Rfl(t)) \rhd (M\, N) : B \mid s \cdot t}$$

These schemes are easily proven to be terminating. However, they are not confluent (take, for instance, the critical pair between the first two reduction schemes and note that it is not joinable). As a consequence we complete these schemes with those in the second group depicted in Fig. 5. The full set of schemes is both confluent and terminating.

Proposition 1. $\curvearrowright$ *is confluent and terminating.*

Termination may be proved automatically by using AProVE [GTSKF04]. Confluence follows by checking local confluence and resorting to Newman's Lemma. We stress that the fact that these reduction schemes are defined over typed terms is crucial for confluence. For example, $Trn(Rfl(s), Rfl(t))$ is typable only in the case that $s = t$.

Definition 1 (Reduction). *Let* $\Longrightarrow$ *stand for permutation reduction to (the unique) normal form. Reduction* ($\to$) *is defined over terms in permutation-reduction normal form as* $\mapsto \circ \Longrightarrow$.

We now address safety of reduction w.r.t. the type system. This involves proving SR and Progress. SR follows from the fact that the reduction schemes originate from proof normalisation. The exception are the second group of schemes of Fig. 5 for which type preservation may also be proved seperately.

[4] Type decorations in equivalence witnesses omitted for the sake of readability.

$$\begin{array}{llll}
\mathcal{D} ::= \Box \mid \lambda a : A.\mathcal{D} \mid \mathcal{D}\ M \mid M\ \mathcal{D} & & \mathcal{C} ::= \Box \mid \lambda a : A.\mathcal{C} \mid \mathcal{C}\ M \mid M\ \mathcal{C} \\
\quad \mid \ \mathsf{let}\, u : A[\Sigma] = \mathcal{D}\,\mathsf{in}\ M & & \quad \mid \ \mathsf{let}\, u : A[\Sigma] = \mathcal{C}\,\mathsf{in}\ M \\
\quad \mid \ \mathsf{let}\, u : A[\Sigma] = M\,\mathsf{in}\ \mathcal{D} \mid !_e^{\Sigma}\mathcal{D} \mid e \rhd \mathcal{D} & & \quad \mid \ \mathsf{let}\, u : A[\Sigma] = M\,\mathsf{in}\ \mathcal{C} \\
\quad \mid \ \alpha\{c_1/M_1, \ldots, c_j/M_j, c_{j+1}/\mathcal{D}, \ldots\} & & \quad \mid \ e \rhd \mathcal{C}
\end{array}$$

$$\begin{array}{rcl}
(\lambda a : A.M)\ N & \xrightarrow{f}_{\beta} & \beta(a^A.s, t) \rhd M^a_{N,t} \\
\mathsf{let}\, u : A[\Sigma] = !_e^{\Sigma} M\,\mathsf{in}\ N & \xrightarrow{f}_{\beta_\Box} & \beta_\Box(u^{A[\Sigma]}.t, \Sigma.s) \rhd N^u_{\Sigma.(M,s,e)} \\
!_e^{\Sigma}\mathcal{C}[\alpha\theta] & \xrightarrow{f}_{\mathcal{L}} & !_e^{\Sigma}\mathcal{C}[\mathit{Trl}(\theta^w, \alpha) \rhd e\theta] \\
\multicolumn{3}{c}{M \xrightarrow{f} N \text{ implies } \mathcal{D}[M] \overset{f}{\mapsto} \mathcal{D}[N]}
\end{array}$$

Fig. 6. Full principle reduction

Proposition 2 (Subject Reduction). *$\Delta; \Gamma; \Sigma \vdash M : A \mid s$ and $M \to N$ implies $\Delta; \Gamma; \Sigma \vdash M : A \mid s$.*

Before addressing Progress we introduce some auxiliary notions. A term is *look-up-blocked* if it is of the form $\mathcal{F}[\alpha\theta_V]$. A term M is *tv-closed* if $\mathsf{fvT}(M) = \mathsf{fvV}(M) = \emptyset$. It is *closed* if it is tv-closed and $\mathsf{fvTrl}(M) = \emptyset$.

Lemma 5 (Canonical forms). *Assume $\cdot; \cdot; \Sigma \vdash V : A \mid s$. Then (1) if $A = A_1 \supset A_2$, then $V = \lambda a : A_1.M$ for some a, M; and (2) if $A = [\![\Sigma'.t]\!]A_1$, then $V = !_e^{\Sigma'} V'$ for some e, V'.*

Proposition 3. *Suppose M is in permutation reduction-normal form, is typable and tv-closed. Then (1) M is a value or; (2) there exists N s.t. $M \mapsto N$ or; (3) M is look-up-blocked.*

Since a closed term cannot be look-up-blocked:

Corollary 1 (Progress). *Suppose M is in permutation reduction normal form, is typable and closed. Then either M is a value or there exists N s.t. $M \to N$.*

5 Strong Normalisation

Full reduction is defined as the union of full principle reduction ($\overset{f}{\mapsto}$, Fig. 6) and permutation reduction ($\curvearrowright$). We address strong normalisation (SN) of a restriction of full reduction, a result which entails SN of a similar restriction of $\lambda^{\natural}$. The restriction consists in requiring that M in the principle reduction axiom $\beta_\Box$ not have occurrences of the audited computation unit constructor "!". In the sequel, we write $\overset{rf}{\mapsto}$ for this restricted notion of reduction.

We first note that $\overset{f}{\mapsto}_{\beta,\beta_\Box}$ is SN. This can be proved by defining a translation $\mathcal{S}(\bullet)$ on $\lambda^{\natural}$ types that "forget" the modal connective and a similar translation from terms in $\lambda^{\natural}$ to terms of the simply typed lambda calculus (with constants) such that: (1) it preserves typability; and (2) it maps full reduction to reduction in the simply typed lambda calculus. Since we already know that $\curvearrowright$ is SN and that reduction in the simply typed lambda calculus is SN, our first result reads:

Proposition 4. $\overset{f}{\mapsto}_{\beta,\beta_\Box} \cup \curvearrowright$ *is SN.*

Therefore, an infinite $\overset{f}{\mapsto} \cup \curvearrowright$ reduction sequence must include an infinite number of $\overset{f}{\mapsto}_{\mathcal{L}}$ steps. Next we show that for $\overset{rf}{\mapsto}$ this is not possible. More precisely, we show that in an infinite $\overset{rf}{\mapsto} \cup \curvearrowright$ reduction sequence, there can only be a finite number of $\overset{f}{\mapsto}_{\mathcal{L}}$ steps. This entails:

Proposition 5. $\overset{rf}{\mapsto} \cup \curvearrowright$ *is SN. Hence $\lambda^{\natural}$, with the same restriction, is SN.*

We now address the proof of the main lemma on which Prop. 5 relies (Lem. 7). We introduce weight functions which strictly decrease by each application of a $\overset{f}{\mapsto}_{\mathcal{L}}$-step and which decreases with each application of a $\overset{rf}{\mapsto}_{\beta,\beta_\Box}$-step or $\curvearrowright$-step. A word on notation: $\langle\!\langle\ \rangle\!\rangle$ is the empty multiset; $\uplus$ is multiset union; and $n \uplus \mathcal{M}$ is the union of the multiset $\langle\!\langle n \rangle\!\rangle$ and $\mathcal{M}$, for $n \in \mathbb{N}$. We use the standard multiset extension $\prec$ of the well-founded ordering $<$ on natural numbers which is also well-founded. For each $n \in \mathbb{N}$ we define $\mathcal{W}_n(M)$ as the multiset given by the following inductive definition on M:

$$\begin{aligned}
\mathcal{W}_n(a) &\overset{\text{def}}{=} \langle\!\langle\ \rangle\!\rangle \\
\mathcal{W}_n(\lambda a : A.M) &\overset{\text{def}}{=} \mathcal{W}_n(M) \\
\mathcal{W}_n(M\,N) &\overset{\text{def}}{=} \mathcal{W}_n(M) \uplus \mathcal{W}_n(N) \\
\mathcal{W}_n(\langle u;\sigma\rangle) &\overset{\text{def}}{=} \langle\!\langle\ \rangle\!\rangle
\end{aligned}
\qquad
\begin{aligned}
\mathcal{W}_n(!^{\Sigma}_{e} M) &\overset{\text{def}}{=} n * \mathcal{W}^t(M) \uplus \\
&\quad \uplus \mathcal{W}_{n*\mathcal{W}^t(M)}(M) \\
\mathcal{W}_n(\mathsf{let}\, u = M \,\mathsf{in}\, N) &\overset{\text{def}}{=} \mathcal{W}_n(M) \uplus \mathcal{W}_n(N) \\
\mathcal{W}_n(\alpha\theta) &\overset{\text{def}}{=} \biguplus_{i\in 1..10} \mathcal{W}_n(\theta(c_i)) \\
\mathcal{W}_n(e \rhd M) &\overset{\text{def}}{=} \mathcal{W}_n(M)
\end{aligned}$$

where $\mathcal{W}^t(M)$ is the number of free trail variables in M plus 1. Note that $\mathcal{W}^t(e \rhd M) \overset{\text{def}}{=} \mathcal{W}^t(M)$. The weight functions informally count the number of trail variables that are available for look-up in audited computation units. The principle reduction axiom β either erases the argument N or substitutes exactly one copy, given the affine nature of truth hypothesis. However, multiple copies of M can arise from $\beta_\Box$ reduction (cf. Fig. 6), possibly under "!" constructors (hence our restriction in item 2 below). Finally, we must take into account that although an trail variable is consumed by $\mathcal{L}$ it also copies the terms in θ (which may contain occurrences of the "!" constructor). In contrast to $\beta_\Box$ however, the consumed trail variable can be used to make the copies of "!" made by $e\theta$ weigh less than the outermost occurrence of "!" on the left-hand side of $\mathcal{L}$.

Lemma 6. *1. $\mathcal{W}_n((\lambda a : A.M)\,N) \succeq \mathcal{W}_n(M^{a}_{N,t})$.*
2. If M has no occurrences of the modal term constructor, then $\mathcal{W}_n(\mathsf{let}\, u : A[\Sigma] = !^{\Sigma}_{e} M \,\mathsf{in}\, N) \succ \mathcal{W}_n(\beta_\Box(u^{A[\Sigma]}.t, \Sigma.s) \rhd N^{u}_{\Sigma.(M,s,e)})$.
3. $\mathcal{W}_n(!^{\Sigma}_{e}\mathcal{C}[\alpha\theta]) \succ \mathcal{W}_n(!^{\Sigma}_{e}\mathcal{C}[\mathit{Trl}(\theta^{w}, \alpha) \rhd e\theta])$.

From these results follow:

Lemma 7. *(1) $M \overset{rf}{\mapsto}_{\beta,\beta_\Box} N$ implies $\mathcal{W}_n(M) \succeq \mathcal{W}_n(N)$; (2) $M \overset{f}{\mapsto}_{\mathcal{L}} N$ implies $\mathcal{W}_n(M) \succ \mathcal{W}_n(N)$; and (3) $M \curvearrowright N$ implies $\mathcal{W}_n(M) = \mathcal{W}_n(N)$.*

6 Example of History Based Access Control

We can also model other phenomena such as Abadi and Fournet's [AF03] mechanism for access control based on execution history. A top-level function declaration is an expression of the form $\mathbf{f} \doteq M$ where $\Delta; \Gamma; \Sigma \vdash M : A \mid s$ together with a typing scheme (left) and evidence equivalence scheme (right):

$$\frac{}{\Delta; \Gamma; \Sigma \vdash \mathbf{f} : A \mid \mathbf{f}}\ \mathsf{TFunc} \qquad \frac{\Delta; \Gamma; \Sigma \vdash A \mid s}{\Delta; \Gamma; \Sigma \vdash \mathsf{Eq}(A, s, \mathbf{f}) \mid \delta_{\mathbf{f}}(s)}\ \mathsf{EqFunc} \tag{1}$$

Also, we have the principle contraction in which the derivation on the left of (1) contracts to:

$$\frac{\Delta; \Gamma; \Sigma \vdash M : A \mid s \qquad \dfrac{\Delta; \Gamma; \Sigma \vdash A \mid s}{\Delta; \Gamma; \Sigma \vdash \mathsf{Eq}(A, s, \mathbf{f}) \mid \delta_{\mathbf{f}}(s)}\ \mathsf{EqFunc}}{\Delta; \Gamma; \Sigma \vdash M : A \mid \mathbf{f}}\ \mathsf{Eq}$$

If $\mathbf{f} \doteq !^{\boldsymbol{\alpha}}_{e} \lambda \boldsymbol{a} : \boldsymbol{A}.M$, then we abbreviate $\mathsf{let}\, u = \mathbf{f} \,\mathsf{in}\, \langle u; \boldsymbol{\alpha}/\boldsymbol{\beta} \rangle\, \boldsymbol{N}$ with $\mathbf{f}\, \boldsymbol{\beta}\, \boldsymbol{N}$. Consider the following top-level declarations:

$$\begin{array}{llll}
\mathbf{delete} \doteq !^{\alpha_d}_{\mathit{Rfl}(q)} \lambda a.\mathsf{if}\ \mathit{FileIOPerm} \in \theta\alpha_d & & \mathbf{cleanup} & \doteq !^{\alpha_c}_{\mathit{Rfl}(r)} \lambda a.\mathbf{delete}\ \alpha_c\ a; \\
\qquad\qquad\qquad\qquad \mathsf{then}\ \mathbf{Win32Delete}\, a & & \mathbf{bad} & \doteq !^{\alpha_b}_{\mathit{Rfl}(s)} \mathbf{cleanup}\ \alpha_b \\
\qquad\qquad\qquad\qquad \mathsf{else}\ \mathbf{securityException}; & & & \quad\text{“}..\backslash \mathit{passwd}'';
\end{array}$$

where the definition of θ requires we first define *perms*, a function assigning a set of (static) permissions to top-level functions: $\mathit{perms}(\mathbf{bad}) \overset{\text{def}}{=} \emptyset$, $\mathit{perms}(\mathbf{cleanup}) \overset{\text{def}}{=} \{\mathit{FileIOPerm}\}$ and $\mathit{perms}(\mathbf{delete}) \overset{\text{def}}{=} \{\mathit{FileIOPerm}\}$:

$$\begin{array}{rlrl}
\theta(\mathit{Rfl}) = \theta(\mathit{Trl}) & \overset{\text{def}}{=} \emptyset & \theta(\mathit{Rpl}) & \overset{\text{def}}{=} \lambda \overline{a} : \mathbb{N}.a_1 \cap .. \cap a_{10} \\
\theta(\mathit{Sym}) = \theta(\mathit{Abs}) & \overset{\text{def}}{=} \lambda a : \mathbb{N}.a & \theta(\beta) = \theta(\beta_\Box) & \overset{\text{def}}{=} \emptyset \\
\theta(\mathit{Trn}) = \theta(\mathit{App}) = \theta(\mathit{Let}) & \overset{\text{def}}{=} \lambda a : \mathbb{N}.\lambda b : \mathbb{N}.a \cap b & \theta(\delta_{\mathbf{f}}) & \overset{\text{def}}{=} \{\mathit{perms}(\mathbf{f})\}
\end{array}$$

Then evaluation of the term $!^{\alpha}_{\mathit{Rfl}(s)}\mathbf{bad}\ \alpha$ will produce a security exception since $\delta_{\mathbf{bad}}(s')$ occurs in the trail consulted by **delete**, for some s'. This term is based on the first example of Sec.4.1 in [AF03]. The second example of that same section (illustrating a case where stack inspection falls short and history based access control has advantages) consists in adding the top-level declaration $\mathbf{badTempFile} \doteq \text{“}..\backslash \mathit{passwd}''$ and extending *perms* by declaring $\mathit{perms}(\mathbf{badTempFile}) \overset{\text{def}}{=} \emptyset$. Then evaluation of $!^{\alpha}_{\mathit{Rfl}(s)}\mathbf{delete}\ \alpha\ \mathbf{badTempFile}$ will also produce a security exception.

7 Conclusions

We have presented a proof theoretical analysis of a functional computation model that keeps track of its computation history. A Curry-de Bruijn-Howard isomorphism of an affine fragment of Artemov's Justification Logic yields a lambda calculus $\lambda^{\natural}$ which models audited units of computation. Reduction in these units generates audit trails that are confined within them. Moreover, these units may look-up these trails and make decisions based on them. We prove type safety for $\lambda^{\natural}$ and strong normalisation for a restriction of it. It would be nice to lift the restriction in the proof of strong normalisation that M in the principle reduction axiom $\beta_{\Box}$ not have occurrences of the audited computation unit constructor "!". Also, it would make sense to study audited computation in a classical setting where, based on audit trail look-up, the current continuation could be disposed of in favour of a more judicious computation. Finally, although examples from the security domain seem promising more are needed in order to better evaluate the applicability of these ideas.

Acknowledgements. To Peter Thiemann for fruitful discussions.

References

[AA01] Alt, J., Artemov, S.: Reflective λ-calculus. In: Kahle, R., Schroeder-Heister, P., Stärk, R.F. (eds.) PTCS 2001. LNCS, vol. 2183, p. 22. Springer, Heidelberg (2001)

[AB07] Artemov, S., Bonelli, E.: The intensional lambda calculus. In: Artemov, S., Nerode, A. (eds.) LFCS 2007. LNCS, vol. 4514, pp. 12–25. Springer, Heidelberg (2007)

[AF03] Abadi, M., Fournet, C.: Access control based on execution history. In: NDSS. The Internet Society (2003)

[Art95] Artemov, S.: Operational modal logic. Technical Report MSI 95-29, Cornell University (1995)

[Art01] Artemov, S.: Explicit provability and constructive semantics. Bulletin of Symbolic Logic 7(1), 1–36 (2001)

[Art08] Artemov, S.: Justification logic. In: Hölldobler, S., Lutz, C., Wansing, H. (eds.) JELIA 2008. LNCS (LNAI), vol. 5293, pp. 1–4. Springer, Heidelberg (2008)

[Bar92] Barendregt, H.: Lambda Calculi with Types. In: Handbook of Logic in Computer Science. Oxford University Press, Oxford (1992)

[BF09] Bonelli, E., Feller, F.: The logic of proofs as a foundation for certifying mobile computation. In: Artemov, S., Nerode, A. (eds.) LFCS 2009. LNCS, vol. 5407, pp. 76–91. Springer, Heidelberg (2008)

[Bre01] Brezhnev, V.: On the logic of proofs. In: Proceedings of the Sixth ESSLLI Student Session, pp. 35–45 (2001)

[DP96] Davies, R., Pfenning, F.: A modal analysis of staged computation. In: 23rd POPL, pp. 258–270. ACM Press, New York (1996)

[DP01a] Davies, R., Pfenning, F.: A judgmental reconstruction of modal logic. Journal of MSCS 11, 511–540 (2001)

[DP01b] Davies, R., Pfenning, F.: A modal analysis of staged computation. Journal of the ACM 48(3), 555–604 (2001)

[GTSKF04] Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Automated termination proofs with aprove. In: van Oostrom, V. (ed.) RTA 2004. LNCS, vol. 3091, pp. 210–220. Springer, Heidelberg (2004)

[ML96] Martin-Löf, P.: On the meaning of the logical constants and the justifications of the logical laws. Nordic J. of Philosophical Logic 1(1), 11–60 (1996)

[NPP08] Nanevski, A., Pfenning, F., Pientka, B.: Contextual modal type theory. ACM Trans. Comput. Log. 9(3) (2008)

A Class of Greedy Algorithms and Its Relation to Greedoids

Srinivas Nedunuri[1], Douglas R. Smith[2], and William R. Cook[1]

[1] Dept. of Computer Science, University of Texas at Austin
{nedunuri,wcook}@cs.utexas.edu
[2] Kestrel Institute, Palo Alto
smith@kestrel.edu

Abstract. A long-standing problem in algorithm design has been to characterize the class of problems for which greedy algorithms exist. Many greedy problems can be described using algebraic structures called matroids, which were later generalized to greedoids. Once in this form, the original problem can be solved using Edmonds' Greedy Algorithm. However there are several practical problems with greedy solutions that either do not have a greedoid representation (e.g. Activity Selection) or for which none is known (e.g. Huffman Coding). This paper presents a new characterization of greedy problems that is strictly more general than greedoids, in that it includes all greedoids, as well as problems such as Activity Selection and Huffman Coding. Unlike matroids, our characterization is an axiomatization of a form of Branch and Bound Search, where greediness is associated with the existence of an appropriate dominance relation. Starting from a definition of optimality of the required solution we derive a recurrence relation. This recurrence can then be transformed into a correct-by-construction program that solves problems in our greedy class, analogous to the Greedy Algorithm.

1 Introduction

A greedy algorithm repeatedly makes a locally optimal choice. For some problems this can lead to a globally optimal solution. In addition to developing individual greedy algorithms, there has been long-term interest in finding a general characterization of greedy algorithms that highlights their common structure. Edmonds [Edm71] characterized greedy algorithms in terms of *matroids*. In 1981, Korte and Lovasz generalized matroids to define *greedoids*, [KLS91]. The question of whether a greedy algorithm exists for a particular problem reduces to whether there exists a translation of the problem into a matroid/greedoid. However, there are several problems for which a matroid/greedoid formulation either does not exist or is very difficult to construct. For example, no known greedoid formulations exist for problems such as Huffman Prefix-free encoding or Activity Selection, [CLRS01].

An alternative approach to constructing algorithms is to take a very general program schema and specialize it with problem-specific information. The result can be a very efficient algorithm for the given problem, [SPW95, SW08, NC09]. One such class of algorithms, Global Search [Smi88], operates by controlled search, where at each level in the search tree there are a number of choices to be explored. Under certain conditions, this collection of choices reduces to a single locally optimal choice, which is the essence of a greedy algorithm. In this paper we axiomatically characterize those conditions. We call our specialization of Global Search *Greedy Global Search (GGS)*. We also show that this characterization of greedy algorithms generalizes greedoids, and therefore also matroids. Our proof does not rely on any particular algorithm, such as the greedy algorithm, but is based solely on the properties of greedoid theory and GGS theory. Finally, we derive a recurrence equation from the statement of correctness of GGS which can be transformed into an executable program through correctness-preserving program transformations. Such a program plays the same role for GGS theory as the Greedy Algorithm does for greedoids. In a companion paper [NSC10] we show how to use our greedy class to systematically derive greedy algorithms.

2 Background

2.1 Specifications and Morphisms

We briefly review some of the standard terminology and definitions from algebra. A *signature* $\Sigma = (S, \mathcal{F})$ consists of a set of sort symbols S and a family $\mathcal{F} = \{F_{v,s}\}$ of finite disjoint sets indexed by $S^* \times S$, where $F_{v,s}$ is the set of operation symbols of rank (v, s). We write $f : v \to s$ to denote $f \in F_{v,s}$ for $v \in S^*, s \in S$ when the signature is clear from context. For any signature Σ the Σ-terms are inductively defined in the usual way as the well-sorted composition of operator symbols and variables. A Σ-formula is a boolean valued term built from Σ-terms and the quantifiers $\forall$ and $\exists$. A Σ-sentence is a closed Σ-formula. A *specification* $T = \langle S, \mathcal{F}, A\rangle$ comprises a signature $\Sigma = (S, \mathcal{F})$ and a set of Σ-sentences A called *axioms*. The generic term *expression* is used to refer to a term, formula, or sentence. A specification $T' = \langle S', \mathcal{F}', A'\rangle$ *extends* $T = \langle S, \mathcal{F}, A\rangle$ if $S \subseteq S', F_{v,s} \subseteq F'_{v,s}$ for every $v \in S^*, s \in S$, and $A \subseteq A'$. Alternatively, we say that T' is an extension of T. A model for T is a structure for $(S, \mathcal{F})$ that satisfies the axioms.We shall use modus ponens, substitution of equals/equivalents, and other natural rules of inference in T. The *theory* of T is the set of sentences closed under the rules of inference from the axioms of T. We shall sometimes loosely refer to T as a theory. A sentence s is a *theorem* of T, written $T \vdash s$ if s is in the theory of T.

A *signature morphism* $f : (S, \mathcal{F}) \to (S', \mathcal{F}')$ maps S to S' and $\mathcal{F}$ to $\mathcal{F}$' such that the ranks of operations are preserved. A signature morphism extends in a unique way to a translation of expressions (as a homomorphism between term algebras) or sets of expressions. A *specification morphism* is a signature morphism that preserves theorems. Let $T = \langle S, \mathcal{F}, A\rangle$ and $T' = \langle S', \mathcal{F},' A'\rangle$ be specifications and let $f : (S, \mathcal{F}) \to (S', \mathcal{F}')$ be a signature morphism between them. f is a specification morphism if for every axiom $a \in A$, $f(a)$ is a theorem of T', ie. $T' \vdash f(a)$. It follows that a specification morphism translates theorems of the source specification to theorems of the target specification. The semantics of a specification morphism is given by a model construction: If $f : T \to T'$ is a specification morphism then every model $\mathcal{M}'$ of T' can be made into a model of T by simply "forgetting" some structure of $\mathcal{M}'$. We say that T' *specializes* T. Practically, this means that any problem that can be expressed in T' can be expressed in T.

It is convenient to generalize the definition of signature morphism slightly to allow the translations of operator symbols to be expressions in the target specification and the translations of sort symbols to be constructions (e.g. products) over the target sorts. A symbol-to-expression morphism is called an *interpretation,* notated $i : T \Rightarrow T'$ where T and T' are the source and target resp. of the morphism.

2.2 Matroids and Greedoids

Matroids date back to the work of Whitney in the 1930's. Greedoids are a generalization of matroids proposed by Korte and Lovasz, [KLS91]. Both have been extensively studied as important algebraic structures with applications in a variety of areas, [BZ92]. Underlying both structures is the notion of a set system:

Definition 1. A *set system* is a pair $\langle S, \mathcal{I} \rangle$ where S is a finite nonempty set and $\mathcal{I}$ is a nonempty collection of subsets of S

A matroid introduces constraints on $\mathcal{I}$:

Definition 2. A *matroid* is a set system $\langle S, \mathcal{I} \rangle$, where the elements of $\mathcal{I}$ are called the *independent* subsets, satisfying the following axioms:

Hereditary. $\forall Y \in \mathcal{I}, \forall X \subseteq Y.\ X \in \mathcal{I}$
Exchange. $\forall X, Y \in \mathcal{I}.\ \|X\| < \|Y\| \Rightarrow \exists a \in Y - X.\ X \cup \{a\} \in \mathcal{I}$

The Hereditary axiom requires that every subset of an independent set is also independent. The Exchange axiom implies that all maximal (ordered by $\subseteq$) independent sets are the same size. Such sets are called *bases.* The classic example of a matroid (and indeed the inspiration for matroids) is the set of independent vectors ($\mathcal{I}$) in a vector space (S). Another example is the collection of acyclic subgraphs ($\mathcal{I}$) of a an undirected graph (S). By associating a weight function $w{:}S \to Nat$ assigning a weight to each item in S, there is a Greedy Algorithm [Edm71] that will compute a (necessarily maximal) weighted independent set $z^* \in \mathcal{I}$, i.e. z^* such that $z^* \in \mathcal{I} \wedge (\forall z' \in \mathcal{I}.\ c(x, z^*) \geq c(x, z'))$ where $c(z) = \sum_{i \in z} w(i)$.

Greedoids [KLS91] are a generalization of matroids in which the Hereditary axiom $\forall Y \in \mathcal{I}, \forall X \subseteq Y.\ X \in \mathcal{I}$ is replaced with a weaker requirement called *Accessibility.*

Definition 3. A *greedoid* is a set system $\langle S, I \rangle$, where the elements of $\mathcal{I}$ are called the *feasible* subsets, satisfying the following axioms:

Accessibility. $X \in \mathcal{I}.\ X \neq \emptyset \Rightarrow \exists a \in X.\ X - \{a\} \in \mathcal{I}$

Exchange. $\forall X, Y \in \mathcal{I}.\ \|X\| < \|Y\| \Rightarrow \exists a \in Y - X.\ X \cup \{a\} \in \mathcal{I}$

Remark. The Hereditary and Accessibility axioms are easier to compare if the Hereditary Axiom is written as $\forall X \in \mathcal{I}, \forall a \in X.\ X - \{a\} \in \mathcal{I}$ which can be shown to be equivalent to the original formulation by induction.

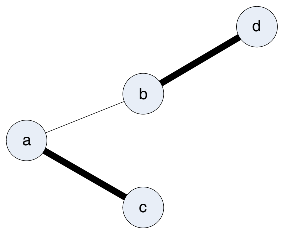


Fig. 1. When the Hereditary axiom does not hold

Why are greedoids important? Consider the problem of finding spanning trees. It is true that given a matroid $\langle S, \mathcal{I}\rangle$ where S is a set of edges forming a connected graph and $\mathcal{I}$ is the set of acyclic subgraphs on that graph, the Greedy Algorithm (see Section 2.4) instantiated on this matroid with an appropriate cost function, is equivalent to Kruskal's algorithm [CLRS01] and returns a minimum spanning tree. However, the collection of *trees* (that is, connected acyclic subgraphs) over a graph does not form a matroid, because the Hereditary Axiom does not hold for a tree. To see this, consider a set system where S is the set of edges $\{(a, b), (a, c), (b, d)\}$ (see Fig. 1) and $\mathcal{I}$ is the set of trees on this graph. Clearly S is feasible but the subset of edges $\{(a, c), (b, d)\}$ is not. However, the weaker Accessibility Axiom does hold, so $\langle S, \mathcal{I}\rangle$ where S is as above, and $\mathcal{I}$ is the set of trees on S forms a greedoid. Instantiated with this greedoid representation of the problem, the Greedy Algorithm is equivalent to Prim's algorithm for MSTs,[CLRS01].

2.3 Greedoid Languages

The implication of the weaker Accessibility axiom for greedoids is that feasible sets should be constructed in an ordered manner, since it is no longer guaranteed that a particular feasible set is reachable from any subset. There is an alternative formulation of greedoids that makes this order explicit [BZ92] which we will utilize. In what follows, a *simple word* over an alphabet S is any word in which no letter occurs more than once and S^*_s is the (finite) set of simple words in S^*. Concatenation of words is denoted in the usual way by concatenation of the corresponding variable names.

Definition 4. A *greedoid language* is a pair $\langle S, \mathcal{L}\rangle$ where S is a finite ground set and $\mathcal{L}$ is a simple language $\mathcal{L} \subseteq S^*_s$ satisfying the following conditions:

Hereditary. $\forall XY \in \mathcal{L}.\ X \in \mathcal{L}$
Exchange. $\forall X, Y \in \mathcal{L}.\ \|X\| < \|Y\| \Rightarrow \exists a \in Y.\ Xa \in \mathcal{L}$

The hereditary and exchange axioms are analogous to the corresponding axioms for matroids, subject to their application to words. That is, the hereditary axiom requires that any prefix of a feasible word is also a feasible. The exchange axiom requires that a shorter feasible word can be extended to a longer feasible one by appending a letter contained in the longer word to the shorter one. As a consequence, all maximal words in $\mathcal{L}$ have the same length.

Bjorner and Ziegler [BZ92] show that the set and language formulations of greedoids are equivalent, that is for every greedoid there is a unique isomorphic greedoid language and v.v. Intuitively, this is because the language version of the greedoid is just enforcing the construction order implied by the feasible set of the greedoid.

2.4 The Greedy Algorithm and Admissible Cost Functions

The greedy algorithm, due to Edmonds [Edm71], is a program schema that is parametrized on a suitable structure such as a matroid or greedoid. Fig. 2 shows the structure of a pseudo-Haskell program for the greedy algorithm that has been parametrized on a greedoid language. First we define the concept of a feasible extension

Definition 5. Given a greedoid language $\langle S, \mathcal{L}\rangle$, the set of *feasible extensions* of a word $A \in \mathcal{L}$, written $ext(A)$ is the set $\{a \mid Aa \in \mathcal{L}\}$.

```
ga(x,y,w) =
    in if exts(ya) = ∅
       then y
       else let m = arbPick(opt(w, exts(ya))) in ga(x,ym,w)
opt(w, s) = {a: ∀a'∈ s.  w(a) >= w(a')}
```

Fig. 2. The Greedy Algorithm parametrized on a Greedoid Language

`arbPick` is a function that picks some element from its argument set. For the the greedy algorithm to be optimal, the cost function must be compatible with the particular structure, or *admissible*. Linear functions are admissible for matroids, but unfortunately not for all greedoids. Admissibility for all greedoids is defined as follows:

Definition 6. Given a greedoid language $\langle S, \mathcal{L}\rangle$, a cost function $c : \mathcal{L} \rightarrow \mathbb{R}$ is *admissible* if, for any $A \in \mathcal{L}$, $a \in ext(A)$, whenever $\forall b \in ext(A).\ c(Aa) \geq c(Ab)$, the following two conditions hold:

$$\forall b \in S, \forall B, C \in S^*.\ ABaC \in \mathcal{L} \wedge ABbC \in \mathcal{L} \Rightarrow c(ABaC) \geq c(ABbC) \quad (2.1)$$

and

$$\forall b \in S, \forall B, C \in S^*.\ AaBbC \in \mathcal{L} \wedge AbBaC \in \mathcal{L} \Rightarrow c(AaBbC) \geq c(AbBaC) \quad (2.2)$$

The first condition states that if a is the best choice immediately after A then it continues to be the best choice. The second condition states that a first and b later is better than b first and a later. A cost function that does not depend on the order of elements in a word immediately satisfies the second condition. Bottleneck functions (functions of the form $\min\{w(X) \mid X \in \mathcal{S}\}$) are an example of admissible functions. Any admissible cost function with a greedoid structure is optimized by the greedy algorithm scheme.

Definition 4 of a greedoid language along with Definition 6of an admissible cost function is what we call *Greedoid Language Theory (GL).*

2.5 Global Search and Problem Specifications

Global Search with Optimality (GSO) is a class of algorithms that operate by controlled search. GSO has an axiomatic characterization as a specification, [Smi88]. In the same way that the greedy algorithm is parametrized on a matroid or greedoid specification, the GSO class has an associated program schema that is parametrized on the GSO specification. We will formalize a specification of GGS that specializes GSO. Before doing so, we will describe a base specification that GSO itself specializes, called an *optimization problem specification* (P).

P is a 6-tuple $\langle D, R, C, i, o, c\rangle$ specifying the problem to be solved. D is the type of the problem input data, R is the result or solution type, $(C, \leq)$ is a well-order that provides some way of measuring the cost of a solution, $i : D \to Boolean$ is an input condition characterizing valid problem inputs over the domain D, $o : D \times R \to Boolean$ is the output condition characterizing *valid* or *feasible* solutions, and $c : D \times R \to C$ is a cost function that returns the cost of a solution. The intent is that a function that solves this problem will take any input $x : D$ that satisfies i and return a $z : R$ that satisfies o for the given x.

A given problem can be classified as an optimization problem by giving an interpretation from the symbols of P to the terms and definitions of the given problem. Here for example is a morphism from P to the Minimum Spanning Tree (MST) problem. The input is a set of edges, where each edge is a pair of nodes with a weight, and nodes are represented by numbers ("!" denotes field access). The result must be a connected acyclic set of edges:

$$\begin{aligned}
D &\mapsto \{Edge\}\\
&\quad Edge = \{a : Node, b : Node, w : Nat\}\\
&\quad Node = Nat\\
R &\mapsto \{Edge\}\\
C &\mapsto Nat\\
i &\mapsto \lambda x.\ true\\
o &\mapsto \lambda x, z.\ connected(z) \wedge acyclic(z)\\
c &\mapsto \lambda(x, z).\ \textstyle\sum_{e \in x} e!w
\end{aligned} \tag{2.3}$$

Appropriate definitions of *connected* and *acyclic* are assumed. Note that an optimal solution to this problem (one that satisfies o and maximizes c) is automatically a spanning tree.

3 Greedy Global Search Theory

Operationally, given a *space* of candidate *solutions* to a given problem (some of which may not be optimal or even feasible solutions), a GGS algorithm partitions (*splits*) the space into *subspaces* (also called *partial solutions*), each of which is recursively searched in turn for optimal solutions. (Such an approach is also the basis of branch-and-bound algorithms, common in AI). At any point, a solution can possibly be extracted from a space, and if correct, compared with the best solution found so far. The process terminates when no space can be further partitioned. The starting point is an *initial space* known to contain all possible solutions to the given problem. The result, if any, is an optimal solution to the problem. Below, we give an axiomatic specification of GGS. The interested reader may refer to Section 3.4 for the associated program schema that is parametrized on this theory.

Sorts. The sorts of a GGS theory are D, R, $\hat{R}$ and C, where D, R, and C are inherited from P, the optimization problem theory, and $\hat{R}$ is the sort of space *descriptors*. A space descriptor is a compact representation of a space and represents all the possible solutions in that space. It is common to make $\widehat{R} = R$.

Operations. In addition to i, o, c which are inherited from P, GGS theory adds additional operators, as befits being a richer theory. As with P, a given problem can be classified as a GGS problem by providing a morphism from the symbols of GGS to the given problem. The operator $\lessdot$ corresponds to the *split* operation mentioned above and χ to the *extract* operation. Note that χ and γ are defined as predicates for uniformity of reasoning in proofs. They are more intuitively thought of as partial functions, one possibly extracting a solution from a space and the other possibly greedily choosing a subspace of a space.

$\hat{z}_0 : D \rightarrow \hat{R}$	initial space
$\in: R \times \hat{R} \rightarrow bool$	is the solution contained in the space?
$\lessdot : D \times \hat{R} \times \hat{R} \rightarrow bool$	is the 1st space a subspace of the 2nd space?
$\chi : R \times \hat{R} \rightarrow bool$	is the solution extractable from the space?
$\gamma : D \times \widehat{R} \times \{\widehat{R}\} \rightarrow bool$	suff. cond. for the space to greedily dominate the set

For ease of reading, ternary operators that take the input x as one of their arguments will from here on be often written in a subscripted infix form. For example,$\gamma(x, \widehat{z}, Z)$ will be written $\widehat{z}\, \gamma_x\, Z$.

Axioms. Finally, the following axioms serve to define the semantics of the operations. $\lessdot^*$ denotes a finite number of applications of the $\lessdot$ operator and is defined as

$$\widehat{s} \lessdot_x^* \widehat{r} = \exists i \geq 0.\ \widehat{s} \lessdot_x^i \widehat{r}$$

where $\widehat{s} \lessdot_x^0 \widehat{r} = (\widehat{r} = \widehat{s})$ and $s \lessdot_x^{i+1} \widehat{r} = \exists \widehat{t} \cdot \widehat{t} \lessdot_x \widehat{r} \wedge \widehat{s} \lessdot_x^i \widehat{t}$. All free variables are universally quantified, and all variables are assumed to have their appropriate type.

A1. $$i(x) \wedge o(x,z) \Rightarrow z \in \widehat{z}_0(x)$$
A2. $$i(x) \Rightarrow (z \in \widehat{y} \Leftrightarrow \exists \widehat{z}.\ \widehat{z} \lessdot_x^* \widehat{y} \wedge \chi(z, \widehat{z}))$$
A3. $$\widehat{z}\, \gamma_x\, ss(x, \widehat{y}) \Rightarrow$$
$$(\exists z \in \widehat{z}, \forall \widehat{z}' \in ss(x, \widehat{y}), \forall z' \in \widehat{z}'.\ o(x, z') \Rightarrow o(x, z) \wedge c(x, z) \geq c(x, z'))$$
A4. $$i(x) \wedge (\exists z \in \widehat{y}.\ o(x, z)) \Rightarrow$$
$$(\exists z^*.\ \chi(z^*, \widehat{y}) \wedge o(x, z^*) \wedge c(x, z^*) = c^*(\widehat{y})) \vee \exists \widehat{z}^* \lessdot_x \widehat{y}.\ \widehat{z}^*\, \gamma_x\, ss(\widehat{y})$$

A1 provides the semantics for the initial space - it states that all feasible solutions are contained in the initial space.

A2 provides the semantics for the subspace operator $\lessdot$ - namely an output object z is in the space denoted by $\widehat{y}$ iff z can be extracted after finitely many applications of $\lessdot$ to $\widehat{y}$. For convenience it is useful to define a function $ss(x, \widehat{y}) = \{\widehat{z} : \widehat{z} \lessdot_x \widehat{y}\}$.

A3 constrains γ to be a *greedy dominance relation.* (Dominance relations have a long history in algorithm development and provide a way of quickly eliminating subspaces that cannot possibly lead to optimal solutions, [Iba77, NC09]). That is, $\widehat{z}\, \gamma_x\, Z$ is sufficient to ensure that $\widehat{z}$ will always lead to at least one feasible solution better than any feasible solution in any space $\widehat{z}'$ in Z. As we will shortly demonstrate, A3 also provides a way of *calculating* the desired γ by a process called *derived antecedents.*

A4 places an additional constraint on γ when applied to the subspaces of $\widehat{y}$: An optimal feasible solution in a space $\widehat{y}$ that contains feasible solutions must be immediately extractable or a subspace of $\widehat{y}$ must greedily dominate the subspaces of $\widehat{y}$. Note that extract is not confined to leaves of the search tree: it is possible that a solution can be extracted from a space that can also be split into subspaces.

Remark. A4 is a little stronger than necessary. In fact, in the case that an optimal feasible solution cannot be immediately extracted from a space, some subspace of that space need only greedily dominate other subspaces in the case that the (parent) space was itself the result of a series of greedy choices. In our experience, weakening A4 in this way would complicate its statement without much of a benefit in practice.

We will show that the class of problems solvable by GGS-theory generalizes the class of problems for which a greedoid representation exists. The way in which this is done is by defining a signature morphism from GGS theory to GL theory, showing the signature morphism is a specification morphism, and then composing that with the isomorphism between GL and G allowing us to conclude that GGS generalizes Greedoids.

3.1 A Signature Morphism from GGS Theory to Greedoid Languages

The signature morphism from GGS to GL is shown in two parts - first the translation of symbols in GGS inherited from P and then the translation of symbols introduced by GGS. Assume the target is a greedoid language $\langle S, \mathcal{L} \rangle$ with associated weight function w and objective function c. The translation of P symbols is[1]: (the [] notation denotes the type of words over an alphabet)

$$\begin{aligned}
D &\mapsto \{S : \{Id\}, \mathcal{L} : \{[Id]\}, w : Id \to C\} \\
R &\mapsto [Id] \\
C &\mapsto Nat \\
i &\mapsto \lambda x.\ \mathit{finite}(x!S) \wedge x!S \neq \emptyset \wedge x.\mathcal{L} \subseteq (x!S)^*_s \wedge x!\mathcal{L} \neq \emptyset \wedge hrd(x!\mathcal{L}) \wedge xch(x!\mathcal{L}) \\
&\quad hrd(\mathcal{L}) = \forall XY \in \mathcal{L}.\ X \in \mathcal{L} \\
&\quad xch(\mathcal{L}) = \forall X, Y \in \mathcal{L}.\ \|X\| < \|Y\| \Rightarrow \exists a \in Y.\ Xa \in \mathcal{L} \\
o &\mapsto \lambda x, z.\ z \in x!\mathcal{L} \\
c &\mapsto c
\end{aligned}$$

The domain D along with the restriction i captures the type of greedoids, and the range R the type of a result, namely some set of objects from the greedoid. The weight of a solution is calculated by c as the sum of the weights of the elements in the solution.

The translation for the additional symbols introduced by GGS is as follows:

$$\begin{aligned}
\widehat{R} &\mapsto [Id] \\
\widehat{z}_0 &\mapsto [] \\
\in &\mapsto \lambda z, \widehat{z}.\ \exists u \in (x.S - \widehat{z})^*.\ z = \widehat{z}u \\
\lessdot &\mapsto \lambda x, \widehat{z}, \widehat{y}.\ \exists a \in x.S - \widehat{y}.\ \widehat{z} = \widehat{y}a \\
\chi &\mapsto \lambda z, \widehat{z}.\ z = \widehat{z} \\
\gamma &\mapsto ?
\end{aligned}$$

To complete the morphism, a translation for γ has to be found, which we will do as part of the process of verifying this morphism is indeed a specification morphism.

3.2 Verifying the Morphism Is a Specification Morphism

To complete the signature morphism and show it is a specification morphism, the translation of the GGS axioms must be provable in GL theory. Axiom A1 holds trivially because the empty list prefixes any list, and A2 can be proved by induction.

A3. To demonstrate A3, we will reason backwards from the consequent. The required assumption will form the greedy dominance relation. We must show the existence of some $z \in \widehat{z}$ for which $\forall \widehat{z}' \in ss_x(\widehat{y}), \forall z' \in \widehat{z}'.\ o(x, z') \Rightarrow o(x, z) \wedge c(x, z) \geq c(x, z')$. We will first show $\forall \widehat{z}' \in ss_x(\widehat{y}), \forall z' \in \widehat{z}', \exists z \in \widehat{z}.\ o(x, z') \Rightarrow o(x, z) \wedge c(x, z) \geq c(x, z')$ and then show the existence of a z that does not depend on $\widehat{z}'$ and z'. Let $\widehat{z} = \widehat{y}a$ for some $a \in x!S - \widehat{y}$, similarly $\widehat{z}' = \widehat{y}u'_1$ for some $u'_1 \in x!S - \widehat{y}$. Now let $z' = \widehat{z}'U'$ for some $U' \in S^*$ be any solution contained in $\widehat{z}'$ (See Fig 3).Reasoning forwards:

[1] This is greedoid theory from an optimization perspective. Of course, other uses of greedoid theory exist.

$o(x, z')$
$= \{\text{unfold defn}\}$
$z' \in x!L$
$= \{\text{abbreviation above}\}$
$\widehat{y}u_1'U' \in x!.L$
$\Rightarrow \{\text{let } U' = U_1'u_2'U_2' \cdots u_n'U_n',\ \text{apply Lemma 1 for } j = 0, \text{ if } a \in ext(\widehat{y})\}$
$\widehat{y}aU_1'u_1'U_2'u_2' \cdots U_{n-1}'u_{n-1}'U_n' \in x!L$
$\Rightarrow \{\text{let } z = \widehat{y}aU_1'u_1'U_2'u_2' \cdots U_{n-1}'u_{n-1}'U_n'\}$
$\exists z \in \widehat{z}.\ o(x, z)$

Next we show that z is better than z'

Under the assumption $a \in ext(\widehat{y}) \wedge \forall a' \in ext(\widehat{y}).\ c(x, \widehat{y}a) \geq c(x, \widehat{y}a')$, the following statements can all be shown: By Lemma 1, and property 2.2,

$$c(x, \widehat{y}aU_1'u_1'U_2'u_2' \cdots U_{n-1}'u_{n-1}'U_n') \geq c(x, \widehat{y}u_1'U_1'aU_2'u_2' \cdots U_{n-1}'u_{n-1}'U_n')$$

and by By Lemma 1, and property 2.2 repeatedly,

$$c(x, \widehat{y}u_1'U_1'aU_2'u_2' \cdots U_{n-1}'u_{n-1}'U_n') \geq c(x, \widehat{y}u_1'U_1'u_2'U_2' \cdots aU_n')$$

and finally by property 2.1

$$c(x, \widehat{y}u_1'U_1'u_2'U_2' \cdots aU_n') \geq c(x, \widehat{y}u_1'U_1'u_2'U_2' \cdots u_n'U_n')$$

and so, by transitivity,

$$c(x, \widehat{y}aU_1'u_1'U_2'u_2' \cdots U_{n-1}'u_{n-1}'U_n') \geq c(x, \widehat{y}u_1'U_1'u_2'U_2' \cdots u_n'U_n')$$

ie. $c(x, z) \geq c(x, z') \Leftarrow a \in ext(\widehat{y}) \wedge \forall a' \in ext(\widehat{y}) \cdot c(x, \widehat{y}a) \geq c(x, \widehat{y}a')$.

We can assert the existence of a single feasible z^* that is better than any feasible z' in $\widehat{z}'$ by taking such a z^* to be the best of every z derived above. Finally, collecting together the assumptions, we get a greedy dominance relation satisfying A3: $\widehat{y}a\, \gamma_x\, \{\widehat{y}a'\} = a \in ext(\widehat{y}) \wedge \forall a' \in ext(\widehat{y}) \cdot c(\widehat{y}a) \geq c(\widehat{y}a)$.

Notation: In what follows, $A - B$, where A and B are words over L, denotes the asymmetric set difference of the two sets A_s and B_s where W_s is the set of symbols contained in the word W, and $\prod_{i=j}^{k} X_i$, for any $X_j, \cdots, X_k \in S^*$, denotes the concatenation $X_j \cdots X_k$. The proof of the lemma is by induction.

Lemma 1. *Given a greedoid $\langle S, L \rangle$, and $Aa \in L, AB \in L$ for some $A, B \in S^*, a \in S$: B can be written $\prod_{i=1}^{n} b_i B_i$ for some $B_1, B_2, \cdots, B_n \in S^*$, such that $\forall j \in [0..n) \cdot A(\prod_{i=0}^{j} b_i B_i)a(\prod_{i=j+1}^{n-1} B_i b_i)B_n \in L$.*

A4. To demonstrate A4 holds, note that if a given word $\widehat{y}$ can be feasibly extended, then, from the greedy dominance relation derived above, there will be a subspace that greedily dominates all subspaces, satisfying the second term of the disjunction. If no such extension exists, a feasible solution can be extracted at any time by taking $\chi(z, \widehat{z}) = (z = \widehat{z})$ and at least one of those will be optimal in $\widehat{z}$, satisfying the first term of the disjunction.

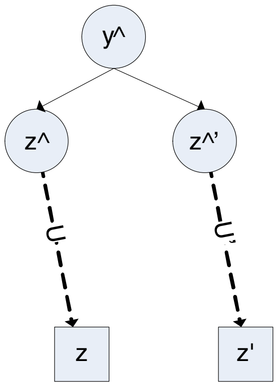


Fig. 3. A solution z in $\hat{z}$ compared with a solution z in $\hat{z}'$

This completes the specialization of GGS by GL.

To show a strict generalization, it is sufficient to demonstrate a problem which can be solved in GGS theory but not using greedoids. One such problem is the Activity Selection Problem [CLRS01],[NSC10]:

> Suppose we have a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed activities that wish to use a resource, such as a lecture hall, which can be used by only one activity at a time. Each activity a_i has a start time s_i and finish time f_i where $0 \leq s_i < f_i < \infty$. If selected, activity a_i takes place in the half-open time interval $[s_i, f_i)$. Activities a_i and a_j are compatible if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. The activity selection problem is to select a maximum-size subset of mutually compatible activities.

The input is a set of activities and a solution is subset of that set. Every activity is uniquely identified by an *id* and a start time (s) and finish time (f). The output condition requires that activities must be chosen from the input set, and that no two activities overlap. The problem specification is:

$$\begin{aligned}
D &\mapsto \{Activity\} \\
&\quad Activity = \{id : Nat, s : Nat, f : Nat\} \\
R &\mapsto \{Activity\} \\
o &\mapsto \lambda(x, z) \cdot noOvp(x, z) \wedge z \subseteq x \\
&\quad noOvp(x, z) = \forall i, j \in z.\ i \neq j \Rightarrow i \preceq j \vee j \preceq i \\
&\quad i \preceq j = i!f \leq j!s \\
c &\mapsto \lambda(x, z).\ \|z\|
\end{aligned}$$

We will now show how the problem can be solved in GGS theory. Most of the types and operators of GGS theory are straightforward to instantiate. We will just set $\widehat{R}$ to be the same as R. The initial space is just the empty set. The subspace relation $\lessdot$ splits a space by selecting an unchosen activity if one exists and adding it to the existing partial solution. The extract predicate χ can extract a solution at any time:

$$\begin{aligned}
\widehat{R} &\mapsto R \\
\widehat{z}_0 &\mapsto \lambda x.\ \emptyset \\
\lessdot &\mapsto \lambda(x, \widehat{z}, \widehat{z}').\ \exists a \in x - \widehat{z} \cdot \widehat{z}' = \widehat{z} \cup \{a\} \\
\chi &\mapsto \lambda(z, \widehat{z}).\ z = \widehat{z}
\end{aligned}$$

$$\begin{aligned}\gamma \mapsto \lambda(x,\widehat{z},Z).\ &\exists \widehat{y}, a \in x.\ Z = ss(\widehat{y}) \wedge \widehat{z} \in Z \wedge \widehat{z} = \widehat{y} \cup \{a\} \wedge \widehat{y} \preceq \{a\}\\ &\wedge \forall (\widehat{y} \cup a') \in Z.\ \widehat{y} \preceq \{a'\} \Rightarrow a!f \leq a'!f\end{aligned}$$

It can be shown that this instantiation satisfies the axioms of GGS theory [NSC10]. To see that the problem cannot be solved with a greedoid representation, consider a set of three activities $\{a_1, a_2, a_3\}$ in which a_1 overlaps with both a_2 and a_3, neither of which overlap each other. Then two feasible solutions are $\{a_1\}$ and $\{a_2, a_3\}$, but neither a_2 nor a_3 can be used to feasibly extend $\{a_1\}$, thus failing to satisfy the Exchange axiom.

Finally, note that another way in which GGS generalizes greedoids is that while the Greedy Algorithm requires an admissible cost function over greedoids, GGS theory places no such restrictions a priori on the cost function.

3.3 A Program Theory for GGS

Starting from a statement of what is desired, namely to compute an optimal feasible solution, we will first formally derive a recurrence, which is then correct by construction. The recurrence can then be transformed into an executable program.Define

$$Fgdy(z,x,\widehat{y}) = z \in opt_c\{z \mid z \in \widehat{y} \wedge o(x,z)\}$$

This is a specification of a function $Fgdy$ to be derived. opt_c is a subset of its argument that is the optimal (w.r.t. the cost function c and the well-order $\geq$), defined as: $\forall z.\ z \in opt_c S = z \in S \wedge (\forall z' \in S.\ c(x,z) \geq c(x,z'))$. In the sequel we will drop the subscript c when it is clear from context.

Theorem 1. *Let $\langle D, R, \widehat{R}, C, i, o, c, \widehat{z}_0, \in, \lessdot, \chi, \gamma\rangle$ be a GGS-Theory as defined above. Then the following characteristic recurrence holds for all x and z:*

$$\begin{aligned}Fgdy(z,x,\widehat{y}) \Leftarrow\ & z \in opt_c\{z \mid\\ & z \in opt_c\{z \mid e(z,\widehat{y}) \wedge o(x,z))\} \vee (\exists \widehat{z} \lessdot \widehat{y}.\ \widehat{z}\,\gamma_x\, ss(x,\widehat{y}) \wedge Fgdy(z,x,\widehat{z}))\}\end{aligned}$$

Proof. (input argument x to o and ss dropped for brevity)

$$\begin{aligned}&Fgdy(z,x,\widehat{y})\\ =\ &\{\text{unfold defn of } Fgdy\}\\ &z \in opt\{z \mid z \in \widehat{y} \wedge o(z))\}\\ =\ &\{\text{provable from A2}\}\\ &z \in opt\{z \mid [\chi(z,\widehat{y}) \vee (\exists \widehat{z}.\ s(x,\widehat{y},\widehat{z}) \wedge z \in \widehat{z})] \wedge o(z)\}\\ =\ &\{\text{distributivity of set comprehension and } opt\}\\ &z \in opt\{z \mid z \in opt\{z \mid \chi(z\widehat{y}) \wedge o(z)\} \vee z \in opt\{z \mid \exists \widehat{z} \in ss(\widehat{y}). z \in \widehat{z} \wedge o(z)\}\}\\ \Leftarrow\ &\{\text{Lemma 2}\}\\ &z \in opt\{z \mid z \in opt\{z \mid \chi(z,\widehat{y}) \wedge o(z)\} \vee (\exists \widehat{z} \lessdot \widehat{y}.\ \widehat{z}\,\gamma_x\, ss(\widehat{y}) \wedge Fgdy(z,x,\widehat{z}))\}\ \square\end{aligned}$$

Lemma 2

$$
\begin{array}{ll}
& opt\{z \mid z \in opt\{z \mid \chi(z,\widehat{y}) \wedge o(z)\} \vee z \in opt\{z \mid \exists \widehat{z} \in ss(\widehat{y}).z \in \widehat{z} \wedge o(z)\}\} \\
\supseteq & \\
& opt\{z \mid z \in opt\{z \mid \chi(z,\widehat{y}) \wedge o(z)\} \vee (\exists \widehat{z} \lessdot \widehat{y}.\ \widehat{z}\, \gamma_x\, ss(\widehat{y}) \wedge Fgdy(z,x,\widehat{z}))\}
\end{array}
$$

Non Triviality. Finally, to demonstrate non-triviality[2] of the recurrence we need to show that if there exists an optimal solution, then one will be found. The following theorem ensures this.

Theorem 2

$$
\begin{array}{l}
(i(x) \wedge \exists z.\ Fgdy(z,x,\widehat{y})) \Rightarrow \exists z \in opt_c\{z \mid \\
z \in opt_c\{z \mid (z,c,\chi(z,\widehat{y}) \wedge o(x,z))\} \vee (\exists \widehat{z} \lessdot \widehat{y}.\ \widehat{z}\, \gamma_x\, ss(x,\widehat{y}) \wedge Fgdy(z,x,\widehat{z}))\}
\end{array}
$$

Proof.

$$
\begin{array}{lc}
 & i(x) \wedge \exists z.\ Fgdy(z,x,\widehat{y}) \\
= & \{\text{defn of } Fgdy\} \\
 & i(x) \wedge \exists z \in opt_c\{z \mid z \in \widehat{y} \wedge o(x,z)\} \\
= & \{\text{property of } opt_c\} \\
 & \exists z.\ i(x) \wedge z \in \widehat{y} \wedge o(x,z) \\
\Rightarrow & \{\text{Axioms A4, A2}\} \\
 & \exists z.\ (\chi(z,\widehat{y}) \vee (\exists \widehat{z} \lessdot \widehat{y}.\ \widehat{z}\, \gamma_x\, ss(x,\widehat{y}) \wedge z \in \widehat{z})) \wedge o(x,z) \\
= & \{\text{distributivity of } \wedge\} \\
 & (\exists z.\ \chi(z,\widehat{y}) \wedge o(x,z)) \vee (\exists z, \widehat{z} \lessdot \widehat{y}.\ z\, \gamma_x\, ss(x,\widehat{y}) \wedge z \in \widehat{z} \wedge o(x,z)) \\
= & \{\text{property of } opt_c\} \\
 & (\exists z. \chi(z,\widehat{y}) \wedge o(x,z)) \vee (\exists z, \widehat{z} \lessdot \widehat{y}. \widehat{z}\, \gamma_x\, ss(x,\widehat{y}) \wedge z \in opt_c\{z \mid z \in \widehat{z} \wedge o(xz)\}) \\
= & \{\text{defn of } Fgdy\} \\
 & (\exists z.\ \chi(z,\widehat{y}) \wedge o(x,z)) \vee (\exists z, \widehat{z} \lessdot \widehat{y}.\ z\, \gamma_x\, ss(x,\widehat{y}) \wedge Fgdy(z,x,\widehat{z})) \\
= & \{\text{property of } opt_c\} \\
 & \exists z \in opt_c\{z \mid \chi(z,\widehat{y}) \wedge o(x,z)\} \vee (\exists z, \widehat{z} \lessdot \widehat{y}.\ \widehat{z}\, \gamma_x\, ss(x,\widehat{y}) \wedge Fgdy(z,x,\widehat{z})) \\
= & \{\text{distributivity of } \exists\} \\
 & \exists z.\ z \in opt_c\{z \mid \chi(z,\widehat{y}) \wedge o(x,z)\} \vee (\exists \widehat{z} \lessdot \widehat{y}.\ \widehat{z}\, \gamma_x\, ss(x,\widehat{y}) \wedge Fgdy(z,x,\widehat{z})) \\
= & \{\text{property of } opt_c\} \\
 & \exists z \in opt_c\{z \mid z \in opt_c\{z \mid \chi(z,\widehat{y}) \wedge o(x,z)\} \\
 & \vee (\exists \widehat{z} \lessdot \widehat{y}.\ \widehat{z}\, \gamma_x\, ss(x,\widehat{y}) \wedge Fgdy(z,x,\widehat{z}))\}
\end{array}
$$

□

3.4 Abstract Program

By the application of correctness preserving transforms, the recurrence proved above can be transformed into the abstract program shown in Alg. 1, written in a pseudo-Haskell style. Further details are in Smith's papers, [Smi88, Smi90].

[2] This is similar but not identical to completeness. Completeness requires that every optimal solution is found by the recurrence, which we do not guarantee.

Algorithm 1. Program Schema for GGS Theory

```
--given x:D satisfying i returns optimal (wrt. cost fn c) z:R satisfying o(x,z)
function solve :: D -> {R}
solve x =
    if $\Phi(\widehat{r}_0$(x)) then (gsolve x $\widehat{r}_0$(x) {}) else {}

function gsolve :: D -> $\widehat{R}$ -> {R} -> {R}
gsolve x space soln =
    let gsubs = {s | s$\in$subspaces x space $\wedge$ $\forall$ss $\in$ subspaces x space,s $\gamma_x$ ss}
        soln' = opt c (soln $\cup${z | $\chi$(z,space) $\wedge$ o(x,z)})
    in if gsubs = {} then soln'
       else let greedy = arbPick gsubs in gsolve x greedy soln'

function opt :: ((D,R) -> C) -> {$\widehat{R}$}-> {$\widehat{R}$}
opt c {s} = {s}
opt c {s,t} = if c(x,s)$>$c(x,t) then {s} else {t}
function subspaces :: D -> $\widehat{R}$-> {$\widehat{R}$}
subspaces x $\widehat{r}$ = {$\widehat{s}$: $\widehat{s} \lessdot_x \widehat{r}\wedge\Phi$(x,$\widehat{s}$)}
```

4 Related Work

Greedoids arose when Korte and Lovasz noticed that the hereditary property required by matroids was stronger than necessary for the Greedy Algorithm of Edmonds to be optimal. However, the exact characterization of the accessible set systems for which the greedy algorithm optimized all linear functions remained an open one until Helman et al. [HMS93] showed that a structure known as a *matroid embedding* was both necessary and sufficient. Matroid embeddings relax the Exchange axiom of greedoids but add two more axioms, so they are simultaneously a generalization and a specialization of greedoids. We have shown that GGS strictly generalizes greedoids.

Curtis [Cur03] has a classification scheme intended to cover *all* greedy algorithms. Unlike Curtis, we are not attempting a complete classification . Curtis also does not relate any of the greedy categories to matroids or greedoids. Finally, Curtis's work is targeted specifically at greedy algorithms but for us greedy algorithms are just a special case of a more general problem of deriving effective global search algorithms. The same work applies to both. In the case that the dominance relation really does not lead to a singleton choice at each split, it can still prove to be highly effective. This was recently demonstrated on some Segment Sum problems we looked at. Although the dominance relation we derived for those problem did not reduce to a strictly greedy choice, it was nonetheless key to reducing the complexity of the search (the width of the search tree was kept constant) and led to a very efficient breadth-first solution that was much faster than comparable solutions derived by program transformation, [NC09].

Another approach has been taken by Bird and de Moor [BM93] who show that under certain conditions a dynamic programming algorithm simplifies into a greedy algorithm. Our characterization can be considered an analogous specialization of branch-and-bound. The difference is that we do not require calculation of the entire program, but specific operators, which is a less onerous task. Also, as pointed out by Curtis [Cur03], the conditions required by Bird and de Moor are not easy to meet.

Charlier [Cha95], also building on Smith's work, proposed a new algorithm class for greedy algorithms that directly embodied the matroid axioms. Using this class, he was able to synthesize Kruskal's MST algorithm and a solution to the $1/1/\sum T_i$ scheduling problem. However he reported difficulty with the equivalent of the Augmentation (comparable to the Exchange) axiom. The difficulty with a new algorithm class is often the lack of a repeatable process for synthesizing algorithms in that class, and this would appear to be what Charlier ran up against. In contrast, by specializing an existing theory (GSO), we can apply all the techniques that are available such as bounds tests, filters, propagators, etc. We are also able to handle a wider class of problems than belong in greedoids.

References

[BM93] Bird, R.S., De Moor, O.: From dynamic programming to greedy algorithms. In: Möller, B., Schuman, S., Partsch, H. (eds.) Formal Program Development. LNCS, vol. 755, pp. 43–61. Springer, Heidelberg (1993)

[BZ92] Björner, A., Ziegler, G.M.: Introduction to greedoids. In: White, N. (ed.) Matroid Applications. Cambridge University Press, Cambridge (1992)

[Cha95] Charlier, B.: The greedy algorithms class: formalization, synthesis and generalization. Technical report (1995)

[CLRS01] Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press, Cambridge (2001)

[Cur03] Curtis, S.A.: The classification of greedy algorithms. Sci. Comput. Program 49(1-3), 125–157 (2003)

[Edm71] Edmonds, J.: Matroids and the greedy algorithm. Math. Programming 1(1), 127–136 (1971)

[HMS93] Helman, P., Moret, B.M.E., Shapiro, H.D.: An exact characterization of greedy structures. SIAM J. on Discrete Math. 6, 274–283 (1993)

[Iba77] Ibaraki, T.: The power of dominance relations in branch-and-bound algorithms. J. ACM 24(2), 264–279 (1977)

[KLS91] Korte, B., Lovasz, L., Schrader, R.: Greedoids. Springer, Heidelberg (1991)

[NC09] Nedunuri, S., Cook, W.R.: Synthesis of fast programs for maximum segment sum problems. In: Intl. Conf. on Generative Programming and Component Engineering (GPCE) (October 2009)

[NSC10] Nedunuri, S., Smith, D.R., Cook, W.R.: Synthesis of greedy algorithms using dominance relations. In: 2nd NASA Symp. on Formal Methods (2010)

[Smi88] Smith, D.R.: Structure and design of global search algorithms. Tech. Rep. Kes.U.87.12, Kestrel Institute (1988)

[Smi90] Smith, D.R.: Kids: A semi-automatic program development system. IEEE Trans. on Soft. Eng., Spec. Issue on Formal Methods 16(9), 1024–1043 (1990)

[SPW95] Smith, D.R., Parra, E.A., Westfold, S.J.: Synthesis of high-performance transportation schedulers. Technical report, Kestrel Institute (1995)

[SW08] Smith, D.R., Westfold, S.: Synthesis of propositional satisfiability solvers. Final proj. report, Kestrel Institute (2008)

On Arithmetic Computations with Hereditarily Finite Sets, Functions and Types

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
tarau@cs.unt.edu

Abstract. Starting from an *executable* "shared axiomatization" of a number of bi-interpretable theories (Peano arithmetic, hereditarily finite sets and functions) we introduce generic algorithms that can be instantiated to implement the usual arithmetic operations in terms of (purely symbolic) hereditarily finite constructs, as well as the type language of Gödel's System **T**. The Haskell code in the paper is available at http://logic.cse.unt.edu/tarau/research/2010/short_shared.hs.

Keywords: formal description of arithmetic and set theoretical data types, hereditarily finite sets and functions, bi-interpretations of Peano arithmetic and finite set theory, symbolic implementations of arithmetic operations, modeling axiomatizations with type classes.

1 Introduction

Natural numbers and finite sets have been used as sometimes competing foundations for mathematics, logic and consequently computer science. The standard axiomatization for natural numbers is provided by Peano arithmetic. Finite set theory is axiomatized with the usual Zermelo-Fraenkel system in which the Axiom of Infinity is replaced by its negation. When the axiom of ϵ-induction, (saying that if properties proven on elements also hold on sets containing them, then they hold for all finite sets) is added, the resulting finite set theory is *bi-interpretable* with Peano arithmetic i.e. they emulate each other accurately through a bijective mapping that allows transporting operations between the two sides ([1]).

This foundational convergence suggests a "shared axiomatization" of Peano arithmetic, hereditarily finite sets and functions, to be used as a unified framework for formally deriving from first principles basic programming language concepts like numbers, sequences and sets.

We develop our "shared axiomatization" described in an *executable form* as a chain of Haskell *type classes* connected by inheritance. Interpretations of the "axiomatized" theories are described as *instances* of the type classes.

The resulting hierarchy of type classes describes incrementally *common computational capabilities* shared by Peano natural numbers, hereditarily finite sets, hereditarily finite functions (sections 2-6) and Gödel's `System T` types (section 7) as well as a total ordering relation (section 5).

While the existence of such a common axiomatization for theories of finite sets and natural numbers can be seen as a consequence of the bi-interpretability results described in [1], our executable specification with Haskell type classes provides unique insights into the shared inductive constructions and ensures that the computational complexity of various operations is asymptotically comparable with that of their usual integer counterparts (as shown in section 8).

2 Sharing Axiomatizations with Type Classes

Haskell's *type classes* [2,3] are a good approximation of axiom systems as they allow one to describe properties and operations generically i.e. in terms of their action on objects of a parametric type. Haskell's type *instances* approximate *interpretations* [1] of such axiomatizations by providing implementations of primitive operations and by refining and possibly overriding derived operations with more efficient equivalents.

We will start by defining a type class that abstracts away commonalities between natural numbers, hereditarily finite sets and hereditarily finite functions.

2.1 The Primitive Operations

The class `SharedAxioms` assumes only a theory of structural equality (as implemented by the class `Eq` in Haskell) and the `Read/Show` superclasses needed for input/output.

An instance of this class is required to implement the following 5 primitive operations:

```
class (Eq n,Read n,Show n)⇒SharedAxioms n where
  e :: n
  o_ :: n→Bool
  o,i,r :: n→n
```

Intuitions for these operations will be provided in the form of several different instances in the next sections, but for now let us just mention that constant function `e` will be interpreted as `0`, as empty set and empty sequence. This type class also endows its instances with generic implementations of the following derived operations:

```
  e_,i_ :: n→Bool
  e_ x = x==e
  i_ x = not (o_ x || e_ x)
```

While not strictly needed at this point, it is convenient also to include in the type class`SharedAxioms` some additional derived operations. We first define an object and a recognizer for what will be interpreted as `1` (and also the sequence `[[]]`), the constant function `u` and the predicate `u_`:

```
  u :: n
  u = o e
  u_ :: n→Bool
  u_ x = o_ x && e_ (r x)
```

Next we implement the successor s and predecessor p functions:

```
  s,p :: n→n

  s x | e_ x = u
  s x | o_ x = i (r x)
  s x | i_ x = o (s (r x))

  p x | u_ x = e
  p x | o_ x = i (p (r x))
  p x | i_ x = o (r x)
```

It is convenient at this point, as we target a diversity of interpretations materialized as Haskell instances, to provide a polymorphic converter between two different instances of the type class SharedAxioms as well as their associated lists, implemented by structural recursion over the representation to convert. The function view allows importing a wrapped object of a different SharedAxioms instance, generically.

```
view :: (SharedAxioms a,SharedAxioms b)⇒a→b
view x | e_ x = e
view x | o_ x = o (view (r x))
view x | i_ x = i (view (r x))
```

A generator for the infinite stream starting with k is obtained using s as follows:

```
allFrom k = k : allFrom (s k)
```

2.2 A Performance Witness *Instance*: Arithmetic as Usual

And for the reader curious by now about how this maps to "arithmetic as usual", here is an instance built around the (arbitrary length) Integer type, also usable as witness on the time/space complexity of our operations.

```
newtype N = N Integer deriving (Eq,Show,Read)

instance SharedAxioms N where
  e = N 0
  o_ (N x) = odd x
  o (N x) = N (2*x+1)
  i (N x) = N (2*x+2)
  r (N x) | x/=0 = N ((x-1) `div` 2)
```

on which one can try out

```
*Shared> (o . i . o) (N 0)
N 9
*Shared> (r . r . r) (N 9)
N 0
```

2.3 The “Reference” *Instance*: Peano Arithmetic

It is important to observe at this point that Peano arithmetic is also an instance of the class `SharedAxioms` i.e. that the class can be used to derive an “axiomatization” for Peano arithmetic through a straightforward mapping of Haskell’s function definitions to axioms expressed in predicate logic. Showing equivalence of behavior with this “reference instance” on various arithmetic operations can be seen as a proof of correctness of their respective algorithms.

```
data Peano = Zero|Succ Peano deriving (Eq,Show,Read)

instance SharedAxioms Peano where
  e = Zero

  o_ Zero = False
  o_ (Succ x) = not (o_ x)

  o x = Succ (twice x) where
    twice Zero = Zero
    twice (Succ x) = Succ (Succ (twice x))

  i x = Succ (o x)

  r (Succ Zero) = Zero
  r (Succ (Succ Zero)) = Zero
  r (Succ (Succ x)) = Succ (r x)
```

And one can also try out, at this point, the polymorphic instance converter `view`:

```
*Shared> view (Succ (Succ Zero)) :: N
N 2
*Shared> view (N 2) :: Peano
Succ (Succ Zero)
*Shared> Succ (view (N 5))
Succ (Succ (Succ (Succ (Succ (Succ Zero)))))
```

3 Computing with Hereditarily Finite Sets

We will now provide an *instance* showing that our “axiomatization” covers the theory of hereditarily finite sets (assuming, of course, that extensionality, comprehension, regularity, ϵ-induction etc. are implicitly provided by type classes like `Eq` and implementation of recursion in the underlying programming language).

Hereditarily finite sets are built inductively from the empty set by adding finite unions of existing sets at each stage. We first define a tree datatype `S`:

```
data S=S [S] deriving (Eq,Read,Show)
```

where the empty set is denoted `S []`. To accurately represent sets, the type `S` would require a type system enforcing constraints on type parameters, saying that all elements covered by the definition are distinct and no repetitions occur in any list of type `[S]`. We will assume this and similar properties of our datatypes, when needed, from now on.

Proposition 1. *Hereditarily finite sets* can do arithmetic *as* instances *of the class* `SharedAxioms` *by implementing successor and predecessor functions* `s` *and* `p`.

```
instance SharedAxioms S where
  e = S []

  o_ (S (S []:_)) = True
  o_ _ = False

  o (S xs) = s (S (map s xs))

  i = s . o
```

Note that the `o` operation is implemented by applying `s` to each branch of the tree. We will now implement `s` and `p` as well as an operation `r` that, as we will see later, reverses the action of both `o` and `i`.

```
  s (S xs) = S (hLift (S []) xs) where
    hLift k [] = [k]
    hLift k (x:xs) | k==x = hLift (s x) xs
    hLift k xs = k:xs

  p (S xs) = S (hUnLift xs) where
    hUnLift ((S []):xs) = xs
    hUnLift (k:xs) = hUnLift (k':k':xs) where k'= p k

  r (S xs) | o_ (S xs) = S (map p ys) where (S ys)=p (S xs)
  r x = r (p x)
```

First note that successor and predecessor operations `s,p` are overridden and that the `r` operation is expressed in terms of `p`, as `o` and `i` were expressed in terms of `s`. Next, note that the `map` combinators and the auxiliary functions `hLift` and `hUnlift` are used to delegate work between successive levels of the tree defining a hereditarily finite set.

To summarize, let us observe that the successor and predecessor operations `s,p` at a given level are implemented through iteration of the same at a lower level and that the "left shift" operation implemented by `o,i` results in initiating `s` operations at a lower level. *Thus the total number of operations is within a constant factor of the size of the trees.*

Finally, let us verify that these operations mimic indeed their more common counterparts on type `N`.

```
*Shared> view (N 42) :: S
S [S [S []],S [S [],S [S []]],S [S [],S [S [S []]]]]
*Shared> p it
S [S [],S [S [],S [S []]],S [S [],S [S [S []]]]]
*Shared> view it :: N
N 41
```

```
*Shared> view (N 5) :: S
S [S [],S [S [S []]]]
*Shared> o it
S [S [],S [S []],S [S [],S [S []]]]
*Shared> view it :: N
N 11
```

A proof by induction that types N and S implement indeed the same successor and predecessor operation as the instance Peano can be carried out with a proof assistant like Coq or ACL2.

Let us note that this implementation of the class SharedAxioms implicitly uses the *Ackermann interpretation* [4] of Peano arithmetic in terms of the theory of hereditarily finite sets, i.e. the natural number associated to a hereditarily finite set is given by the function

$$f(x) = \texttt{if } x = \emptyset \texttt{ then } 0 \texttt{ else } \sum_{a \in x} 2^{f(a)}$$

Let us summarize what's unusual with instance S of the class SharedAxioms: it shows that successor and predecessor operations can be performed with *hereditarily finite sets playing the role of natural numbers*. As natural numbers and finite ordinals are in a one-to-one mapping, this instance shows that *hereditarily finite sets can be seen as finite ordinals* directly, without using the "computationally explosive" von Neumann construction (which defines ordinal n as the set $\{0, 1, \ldots, n-1\}$).

We will now provide an instance defined in terms of a more efficient hereditarily finite construct.

4 Computing with Hereditarily Finite Functions

Hereditarily finite functions, described in detail in [5,6], extend the inductive mechanism used to build hereditarily finite sets to finite functions on natural numbers (conveniently represented as finite sequences i.e. *lists* of natural numbers in Haskell). They are expressed using a similar datatype, denoted F here. The key difference is that, in this case, order is important, and that identical elements can occur at each level. Hereditarily finite functions can also be seen as compressed encodings of hereditarily finite sets, where, at each level, only increments between elements are represented. The first set of operations are similar to the ones on the type S:

```
data F = F [F] deriving (Eq,Read,Show)

instance SharedAxioms F where
  e= F []

  o_ (F (F []:_))=True
  o_ _ = False

  o (F xs) = F (e:xs)
```

```
i = s . o

r (F (x:xs)) | e_ x = F xs
r x = r (p x)
```

The code for `s` and `p` is also similar to the one given for hereditarily finite sets, except that this time `s` and `p` are co-recursive.

```
s (F xs) = F (hinc xs) where
  hinc ([]) = [e]
  hinc (x:xs) | e_ x= (s k):ys where (k:ys)=hinc xs
  hinc (k:xs) = e:(p k):xs

p (F xs) = F (hdec xs) where
  hdec [x] | e_ x= []
  hdec (x:k:xs) | e_ x= (s k):xs
  hdec (k:xs) = e:(hdec ((p k):xs))
```

Proposition 2. *Hereditarily finite functions* can do arithmetic *as* instances *of the class* `SharedAxioms` *by implementing successor/predecessor functions* `s` *and* `p`.

As with the type `S`, the total number of operations is proportional to the size of the trees. Given that `F`-trees are significantly smaller than `S`-trees, various operations will perform significantly faster, as in this representation only "increments" or "decrements" from one subtree to the next are computed (functions `hinc` and `hdec`). One can also observe that parallelization of the algorithm can be achieved by adapting *parallel prefix sum* computations as in [7]. A few examples follow:

```
*Shared> view (N 42) :: S
S [S [S []],S [S [],S [S []]],S [S [],S [S [S []]]]]
*Shared> view (N 42) :: F
F [F [F []],F [F []],F [F []]]
*Shared> s it
F [F [],F [],F [F []],F [F []]]
*Shared> view it :: N
N 43
*Shared> view (N 5) :: F
F [F [],F [F []]]
*Shared> view (o it) :: N
N 11
```

As a side note, let's observe that a *parenthesis language* representation[1] of hereditarily finite functions provides a self delimiting *prefix code* and the *Kraft inequality* holds for any given encoding, as well as, recursively, for each of its "parts". This could make this encoding interesting for both code and data representations in algorithmic information theory [8,9,10].

[1] Obtained, for instance, by deleting all F symbols, spaces and commas in the printed form of a hereditarily finite function.

5 Defining a Total Order

We will define next a well-founded total order relation, matching the natural order induced by repeated application of the successor function.

```
class (SharedAxioms n) ⇒ SharedOrdering n where
```

Let the *length* of an object be the number of `i` and `o` operations used to build it starting from `e` (or, equivalently, the number of applications of `r` needed to reduce it to `e`). Efficient comparison uses the fact that:

Proposition 3. *With our representation only sequences of equal lengths can be equal.*

We start by comparing lengths:

```
  lcmp :: n→n→Ordering

  lcmp x y | e_ x && e_ y = EQ
  lcmp x y | e_ x && not(e_ y) = LT
  lcmp x y | not(e_ x) && e_ y = GT
  lcmp x y = lcmp (r x) (r y)
```

Comparison can now proceed by case analysis, the interesting case being when lengths are equal (function `samelen_cmp`):

```
  cmp :: n→n→Ordering

  cmp x y = ecmp (lcmp x y) x y where
     ecmp EQ x y = samelen_cmp x y
     ecmp b _ _ = b

  samelen_cmp :: n→n→Ordering

  samelen_cmp x y | e_ x && e_ y = EQ
  samelen_cmp x y | e_ x && not(e_ y) = LT
  samelen_cmp x y | not(e_ x) && e_ y = GT
  samelen_cmp x y | o_ x && o_ y = samelen_cmp (r x) (r y)
  samelen_cmp x y | i_ x && i_ y = samelen_cmp (r x) (r y)
  samelen_cmp x y | o_ x && i_ y =
    downeq (samelen_cmp (r x) (r y)) where
      downeq EQ = LT
      downeq b = b
  samelen_cmp x y | i_ x && o_ y =
    upeq (samelen_cmp (r x) (r y)) where
      upeq EQ = GT
      upeq b = b
```

Finally, boolean comparison operators are defined as follows:

```
  lt,gt,eq :: n→n→Bool

  lt x y = LT==cmp x y
  gt x y = GT==cmp x y
  eq x y = EQ==cmp x y
```

After adding the instances

```
instance SharedOrdering N
instance SharedOrdering Peano
instance SharedOrdering S
instance SharedOrdering F
```

one can see that all operations extend naturally:

```
*Shared> lt (N 2009) (N 2010)
True
*Shared> lt (S []) (S [S [],S []])
True
```

The last operation also shows that:

Proposition 4. *We have a well-founded* total order *on hereditarily finite sets without needing the von Neumann ordinal construction used in [1] to complete the bi-interpretation from hereditarily finite sets to natural numbers.*

This replicates a recent result described in [11] where a lexicographic ordering is used to simplify the proof of bi-interpretability of [1].

6 Arithmetic Operations

Our next refinement adds key arithmetic operations in the form of a type class extending `SharedAxioms`. We start with addition (`a`) and difference (`d`):

```
class (SharedOrdering n) ⇒ SharedArithmetic n where
  a,d :: n→n→n

  a x y | e_ x = y
  a x y | e_ y = x
  a x y | o_ x && o_ y =    i (a (r x) (r y))
  a x y | o_ x && i_ y = o (s (a (r x) (r y)))
  a x y | i_ x && o_ y = o (s (a (r x) (r y)))
  a x y | i_ x && i_ y = i (s (a (r x) (r y)))

  d x y | e_ x && e_ y = e
  d x y | not(e_ x) && e_ y = x
  d x y | not (e_ x) && x==y = e
  d z x | i_ z && o_ x = o (d (r z) (r x))
  d z x | o_ z && o_ x = i (d (r z) (s (r x)))
  d z x | o_ z && i_ x = o (d (r z) (s (r x)))
  d z x | i_ z && i_ x = i (d (r z) (s (r x)))
```

Proposition 5. *Addition* `a` *and subtraction* `d` *can be implemented generically, with asymptotic complexity proportional to the size of the operands, for natural numbers, hereditarily finite sets and hereditarily finite functions.*

Next, we define multiplication.

```
  m :: n→n→n  -- multiplication

  m x _ | e_ x = e
  m _ y | e_ y = e
  m x y = s (m0 (p x) (p y)) where
    m0 x y | e_ x = y
    m0 x y | o_ x = o (m0 (r x) y)
    m0 x y | i_ x = s (a y  (o (m0 (r x) y)))

  db,hf :: n→n -- double and half

  db = p . o
  hf = r . s
```

Exponentiation by squaring follows - easier for powers of two (`exp2`), then the general case (`pow`):

```
  exp2 :: n→n -- power of 2

  exp2 x | e_ x = u
  exp2 x = db (exp2 (p x))

  pow :: n→n→n -- power y of x

  pow _ y | e_ y = u
  pow x y | o_ y = m x (pow (m x x) (r y))
  pow x y | i_ y = m (m x x) (pow (m x x) (r y))
```

After defining instances

```
instance SharedArithmetic N
instance SharedArithmetic Peano
instance SharedArithmetic S
instance SharedArithmetic F
```

operations can be tested under various representations

```
*Shared> a (Succ Zero) (Succ Zero)
Succ (Succ Zero)
*Shared> a (N 32) (N 10)
N 42
*Shared> view (N 6) :: F
F [F [F []],F []]
*Shared> m it it
F [F [F [F []]],F [F [F []]]]
*Shared> view it :: N
N 36
*Shared> pow (N 6) (N 10)
N 60466176
*Shared> pow (view (N 6)::F) (view (N 10) ::F)
F [F [F [F []],F [F []]],F [F [F []]],F [F []],F [F []],
  F [F []],F [],F [F [F []]],F [],F []]
*Shared> view it::N
N 60466176
```

7 Computing with Gödel's System T Types

We will show here how our shared axiomatization framework can be extended with a new, somewhat unusual instance, that brings the ability to do arithmetic computations with an important ancestor of modern type systems: Gödel's System **T**.

Definition 1. *In Gödel's System* **T** *[12] a type is either the basic type* t_0 *or* $t_1 \to t_2$ *where* t_1 *and* t_2 *are types.*

The basic type t_0 is usually interpreted as the type of natural numbers. We will show now that natural numbers can be emulated directly with types, by using a single constant T as basic type, (seen as representing 0) and that all the benefits of our shared axiomatization framework (including views as sets or sequences) can be extended to System **T** types.

First, note that, guided by the *Catalan family* isomorphism between rooted ordered trees and rooted ordered binary trees[2] we can bring with a *functor* defined from hereditarily finite sequences to binary trees the definitions of `s`, `p` and `r` into corresponding definitions in the language of System **T** types.

First, we define the data type for System **T** objects:

```
infixr 5 :→

data T = T | T :→ T deriving (Eq, Read, Show)
```

As in the case of hereditarily finite sets and functions we will start by defining the first 5 primitive operations:

```
instance SharedAxioms T where
  e = T

  o_ (T:→_) = True
  o_ x = False

  o x = T :→ x

  i = s . o

  r (T:→y) = y
  r (x:→y) = p (p x:→y)
```

Note that the successor and predecessor functions `s` and `p` are used in the definition of `i` and `r`. We implement them as overrides for `s` and `p`

[2] That manifests itself in languages like Prolog or LISP as the dual view of lists as a representation of sequences or binary CONS-cell trees.

```
  s T = T:→T
  s (T:→y) = s x:→y' where x:→y' = s y
  s (x:→y) = T:→(p x:→y)

  p (T:→T) = T
  p (T:→(x:→y)) = s x:→y
  p (x:→y) = T:→p (p x:→y)
```

Note that, as in the case of similar operations in sections 3 and 4, one could have simply defined two auxiliary functions `s'` and `p'` to be used in implementing `i`, `r` and then reuse the generic definition of `s` and `p` given in section 2, to ensure that the instance `T` matches formally the concept of *interpretation* of our axioms expressed in terms of the 5 primitive operations `e, o, o_, i, r`.

After observing that $z = x :\rightarrow y$ if and only if (under a natural number interpretation) $z = 2^x(2y + 1)$, it can be proven by structural induction that:

Proposition 6. *The Gödel System* **T** *types, as represented by the data type* `T`*, implement the same successor and predecessor operation as the instance* `Peano`*.*

We are ready now to turn System **T** types into everything else (natural numbers, finite sets, finite functions)

```
instance SharedOrdering T
instance SharedArithmetic T
```

We can now try out the stream generator `allOf` providing a recursive enumeration of all types:

```
*Shared> take 7 (allFrom T)
 [ T,
   T :→ T,
   (T :→ T) :→ T,
   T :→ (T :→ T),
   ((T :→ T) :→ T) :→ T,
   T :→ ((T :→ T) :→ T),
   (T :→ T) :→ (T :→ T) ]
```

Arithmetic and set computations, operating directly on types, are now derived automatically:

```
*Shared> | o_ x
*Shared> let three=s two
*Shared> two
(T :→ T) :→ T
*Shared> three
T :→ (T :→ T)
*Shared> m two three
(T :→ T) :→ (T :→ T)
*Shared> view it :: N
N 6
*Shared> pow two (pow three two)
(T :→ (((T :→ T) :→ T) :→ T)) :→ T
```

```
*Shared> view it :: N
N 512
*Shared> pow (N 2) (pow (N 3) (N 2))
N 512
```

8 A Performance Test: The Collatz Conjecture

We now use a variant of the 3x+1 problem / Collatz conjecture / Syracuse function[3] [14] (see also http://en.wikipedia.org/wiki/Collatz_conjecture) to compare the relative performance of various *instances* and to test that our computations perform within their expected complexity bounds. It is easy to show that the Collatz conjecture is true if and only if the function `nsyr`, implementing the n-th iterate of the *Syracuse function*, always terminates:

```
syracuse n = trim (a (m six n) four) where
  four = s (s (s (s e)))
  six = s (s four)

  trim xs | i_ xs = trim (hf xs)
  trim xs  = hf xs

nsyr n | e_ n = [e]
nsyr n = n : nsyr (syracuse n)
```

The first 8 sequences are computed as follows:

```
*Shared> map (nsyr.N) [0..7]
[[N 0],[N 1,N 2,N 0],[N 2,N 0],[N 3,N 5,N 8,N 6,N 2,N 0],
 [N 4,N 3,N 5,N 8,N 6,N 2,N 0],[N 5,N 8,N 6,N 2,N 0],
 [N 6,N 2,N 0],[N 7,N 11,N 17,N 26,N 2,N 0]]
```

The function `sumsyr` forces the evaluation of the function `nsyr` for `k` successive numbers starting from `n` and returns the sum of the length of the outputs i.e. respectively `7995, 18406, 25243.`

```
sumsyr :: (SharedArithmetic a) ⇒ Int → a → Int
sumsyr k n = sum (map (length.nsyr) (take k (allFrom n)))
```

bitsize	T	N	F	S
31	4173	4607	8685	18547
64	15414	17011	31857	113103
97	30854	33761	64060	331107

Fig. 1. Timings for `T, N, F, S` in ms

Fig. 1 shows timings for `sumsyr 100` on views of 123456780 as types `N,T,F,S`, and then the same digits repeated twice and three times, for data provided

[3] Sequence A173732 in [13].

by various instances of `SharedAxioms`. They show low polynomial growth in the bitsize of the inputs for the respective instances. Timings also indicate significant gains for hereditarily finite functions (col. `F`) vs. hereditarily finite sets (col. `S`). Surprisingly, computations on binary trees representing `System T` types (col. `T`) turn out to be slightly faster than those on integers of type `N`.

9 Related Work

The techniques described in this paper originate in the data transformation framework described in [15,5,6,16]. The main new contribution is that while our previous work can be seen as "an existence proof" that, for instance, arithmetic computations can be performed with symbolic objects like hereditarily finite sets, here we show it constructively. Moreover, we lift our conceptual framework to a polymorphic axiomatization which turns out to have as *interpretations* (instances in Haskell parlance) natural numbers, hereditarily finite sets and functions, System **T** types.

Natural number encodings of hereditarily finite sets (that have been the main inspiration for our concept of hereditarily finite functions) have triggered the interest of researchers in fields ranging from Axiomatic Set Theory to Foundations of Logic [17,1,18].

An emulation of Peano and conventional binary arithmetic operations in Prolog, is described in [19]. Their approach is similar as far as a symbolic representation is used. The key difference with this paper is that our operations work on tree structures, and as such, they are not based on previously known algorithms. Our tree-based algorithms are also genuinely parallelizable as shown in [7].

Binary number-based axiomatizations of natural number arithmetic are likely to be folklore, but having access to the the underlying theory of the calculus of constructions [20] and the inductive proofs of their equivalence with Peano arithmetic in the libraries of the `Coq` [21] proof assistant has been particularly enlightening to the author. On the other hand we have not found in the literature any such axiomatizations in terms of hereditarily finite sets or hereditarily finite functions, as described in this paper.

10 Conclusion

We have described, in the form of a literate Haskell program, a few unusual algorithms expressing arithmetic computations in terms of "symbolic structures" like hereditarily finite sets, hereditarily finite functions and System **T** types.

Besides possible practical applications of our algorithms to symbolic and/or arbitrary length integer arithmetic packages, our type classes based "shared axiomatization" technique can be seen as a framework that unifies formal specification of fundamental mathematical concepts in a directly executable form.

References

1. Kaye, R., Wong, T.L.: On Interpretations of Arithmetic and Set Theory. Notre Dame J. Formal Logic 48(4), 497–510 (2007)
2. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad-hoc. In: POPL, pp. 60–76 (1989)
3. Jones, S.P., Jones, M., Meijer, E.: Type classes: An exploration of the design space. In: Haskell Workshop (1997)
4. Ackermann, W.F.: Die Widerspruchsfreiheit der allgemeinen Mengenlhere. Mathematische Annalen (114), 305–315 (1937)
5. Tarau, P.: A Groupoid of Isomorphic Data Transformations. In: Carette, J., Dixon, L., Coen, C.S., Watt, S.M. (eds.) Calculemus 2009. LNCS (LNAI), vol. 5625, pp. 170–185. Springer, Heidelberg (2009)
6. Tarau, P.: An Embedded Declarative Data Transformation Language. In: Proceedings of 11th International ACM SIGPLAN Symposium PPDP 2009, Coimbra, Portugal, September 2009, pp. 171–182. ACM, New York (2009)
7. Misra, J.: Powerlist: a structure for parallel recursion. ACM Transactions on Programming Languages and Systems 16, 1737–1767 (1994)
8. Li, M., Vitányi, P.: An introduction to Kolmogorov complexity and its applications. Springer, New York (1993)
9. Chaitin, G.J.: A theory of program size formally identical to information theory. J. Assoc. Comput. Mach. 22, 329–340 (1975)
10. Calude, C., Salomaa, A.: Algorithmically coding the universe. In: Developments in Language Theory, pp. 472–492. World Scientific, Singapore (1994)
11. Pettigrew, R.: On Interpretations of Bounded Arithmetic and Bounded Set Theory. Notre Dame J. Formal Logic 50(2), 141–151 (2009)
12. Gödel, K.: Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. Dialectica 12(280-287), 12 (1958)
13. Sloane, N.J.A.: The On-Line Encyclopedia of Integer Sequences (2010), published electronically at `http://www.research.att.com/~njas/sequences`
14. Lagarias, J.C.: The 3x+1 Problem: An Annotated Bibliography (1963-1999) (2008), `http://arXiv.org` 0309224v11
15. Tarau, P.: Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell. In: Proceedings of ACM SAC 2009, pp. 1898–1903. ACM, New York (2009)
16. Tarau, P.: Declarative Combinatorics: Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell, 104 pages (January 2009), (unpublished draft) `http://arXiv.org/abs/0808.2953`
17. Takahashi, M.o.: A Foundation of Finite Mathematics. Publ. Res. Inst. Math. Sci. 12(3), 577–708 (1976)
18. Kirby, L.: Addition and multiplication of sets. Math. Log. Q. 53(1), 52–65 (2007)
19. Kiselyov, O., Byrd, W.E., Friedman, D.P., Shan, C.-c.: Pure, declarative, and constructive arithmetic relations (declarative pearl). In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 64–80. Springer, Heidelberg (2008)
20. Coquand, T., Huet, G.: The calculus of constructions. Information and Computation 76(2/3), 95–120 (1988)
21. The Coq development team: The Coq proof assistant reference manual. LogiCal Project. Version 8.0 (2004)

A Modality for Safe Resource Sharing and Code Reentrancy*

Rui Shi[1], Dengping Zhu[2], and Hongwei Xi[3]

[1] Yahoo! Inc.
[2] Bloomberg Inc.
[3] Boston University

Abstract. The potential of linear logic in facilitating reasoning on resource usage has long been recognized. However, convincing uses of linear types in practical programming are still rather rare. In this paper, we present a general design to effectively support practical programming with linear types. In particular, we introduce and then formalize a modality, which we refer to as the sharing modality, in support of sharing of linear resources (with no use of locks). We develop the underlying type theory for the sharing modality and establish its soundness based on a notion of *types with effects*. We also point out an intimate relation between this modality and the issue of code reentrancy. In addition, we present realistic examples to illustrate the use of sharing modality, which are verified in the programming language ATS and thus provide a solid proof of concept.

1 Introduction

Although linear logic arose historically from domain theory [6], its potential in facilitating reasoning on resource usage has been recognized since the very beginning of its invention. For instance, Asperti showed an interesting way to describe Petri nets [1] in terms of linear logic formulas. In type theory, we have so far seen a large body of research on using linear types to facilitate memory management (e.g. [16,4,15,10,8,7,17]).

When programming with linear resources, we often need to *thread these resources through functions*. Suppose that we have a linear array A of type $\mathbf{larray}(T)$, where T is the type for the elements stored in A. In order to use linear arrays, we need the following access functions:

$$\begin{array}{r}
\mathit{lsub} : \mathbf{larray}(T) \otimes \mathbf{int} \rightarrow \mathbf{larray}(T) \otimes T \\
\mathit{lupdate} : \mathbf{larray}(T) \otimes \mathbf{int} \otimes T \rightarrow \mathbf{larray}(T) \otimes \mathbf{1}
\end{array}$$

where we use $\otimes$ for linear conjunction and $\mathbf{1}$ for the unit type. Intuitively, the subscripting function on a *linear* array A needs to take A and an index i and then return both A and the content stored in the cell indexed by i (so that A is still

* This work is partially supported by NSF grants No. CCR-0229480 and No. CCF-0702665.

available for subsequent uses). The same holds for the update function *lupdate* as well. With linearity, it can be guaranteed that there is exactly one access path for each linear resource so that the state of the resource can be soundly reasoned about (e.g. linear arrays can be safely freed). However, the need for threading linear resources can often be burdensome or even impractical (in cases where a large number of resources are involved).

The subscripting function on a *sharable* array[1] as is supported in languages such as ML takes the array and an index and then returns only the content stored in the cell indexed by i. Intuitively, a sharable array is just a linear array that can be shared (in some restricted manner if necessary). However, we find it highly challenging to properly explain in a type theory how a sharable array can be implemented on top of a linear array. In most safe languages, arrays are treated as an abstract data structure. For instance, there is an abstract type constructor **array** in ML that takes a type T to form the type $\mathbf{array}(T)$ for sharable arrays in which the stored elements are of type T, and the following functions are provided for creating (and initializing as well), subscripting and updating arrays, respectively:

$$\begin{array}{rcl} \texttt{array} & : & \mathbf{int} * T \rightarrow \mathbf{array}(T) \\ \texttt{sub} & : & \mathbf{array}(T) * \mathbf{int} \rightarrow T \\ \texttt{update} & : & \mathbf{array}(T) * \mathbf{int} * T \rightarrow \mathbf{1} \end{array}$$

However, the type constructor **array** cannot be defined in ML and the functions `array`, `sub` and `update` have to be implemented in other (unsafe) languages such as C or assembly and then *assumed* to possess the types assigned to them. Though simple and workable, this approach to supporting arrays in safe languages is evidently uninspiring and unsatisfactory when type theory is of the concern. Also, this approach makes it difficult, if not entirely impossible, to directly manipulate memory at low level, which is often indispensable in systems programming.

We can also see the need for sharable data structures from a different angle. As a simple but crude approximation, let us assume that the memory allocation/deallocation functions *malloc* and *free* are assigned the following types:

$$\mathit{malloc} : \mathbf{int} \rightarrow \mathbf{larray(top)} \qquad \mathit{free} : \mathbf{larray(top)} \rightarrow \mathbf{1}$$

where we use **top** for the top type, that is, every type is a subtype of **top**. Clearly, this type assignment for *malloc* and *free* relies on the assumption that the free list[2] used in implementing these functions is shared. Otherwise, *malloc* and *free* need to be assigned the following types:

$$\begin{array}{r} \mathit{malloc} : \mathbf{freelist} \otimes \mathbf{int} \rightarrow \mathbf{freelist} \otimes \mathbf{larray(top)} \\ \mathit{free} : \mathbf{freelist} \otimes \mathbf{larray(top)} \rightarrow \mathbf{freelist} \otimes \mathbf{1} \end{array}$$

where we use **freelist** as a type for the (linear) free list. This simply means that the free list must be threaded through every client program that makes use

[1] Note that a sharable array is a mutable data structure and it should not be confused with a functional array (e.g. based on a Braun tree).

[2] A free list is a data structure commonly used in implementing *malloc* and *free* for the purpose of maintaining a list of available memory blocks.

of either *malloc* or *free*, which makes it rather impractical to construct critical system libraries such as memory allocator in this manner.

Ideally, resources should be manipulated at low-level where linear types are used to reason about resource usage. While, in order to be practical, it is desirable to make linear resources sharable at some point as shown by the above motivating examples. Apparently, a naive treatment which simply turns linear resources to nonlinear ones without any restrictions does not work. If unrestricted access to shared resources were allowed, the presence of alias could easily break type soundness.

In this paper, we develop a type system where linear types are available for safe programming with resources. In order to support safe resource sharing, we introduce a modality that intuitively means *a shared resource of some linear type can be borrowed only if it is guaranteed that another resource of the same linear type is to be returned*. The primary contribution of the paper lies in the identification and then the formalization of this modality through a notion of *types with effects* [9], where some interesting as well as challenging technical issues are addressed. As an application, we demonstrate that various features for safe memory manipulation at low level (including memory allocation/initialization and pointer arithmetic) can be effectively supported in the type system we develop. We show, for example, safe implementations of sharable arrays based on primitive memory operations, and we believe that such implementations are done (as far as we know) for the first time in a programming language. In addition, we also point out an intimate relation between this modality and the issue of code reentrancy, providing a formal account for code reentrancy as well as a means that can prevent non-reentrant functions like *malloc* and *free* from being called reentrantly (e.g., in threads).

The type system we ultimately develop involves a long line of research on dependent types [21,18], linear types [22], programming with theorem proving [3], and type theory for resource sharing. To facilitate understanding, we give a detailed presentation of a simple but rather abstract type system that supports resource sharing, and then outline extensions of this simple type system with advanced types and programming features. The interesting and realistic examples we show all involve dependent types and possibly polymorphic types. In addition, they all rely on the feature of programming with theorem proving.

We organize the rest of the paper as follows. In Section 2, we present a language $\mathcal{L}_0$ with a simple linear type system, setting up some machinery for further development. We extend $\mathcal{L}_0$ to $\mathcal{L}_\Box$ with a modality in Section 3 to address the issue of resource sharing in programming. Furthermore, we extend $\mathcal{L}_\Box$ to $\mathcal{L}_\Box^{\forall,\exists}$ in Section 4 by incorporating universally as well as existentially quantified types, preparing to support direct memory manipulation at low level. In Section 5, we demonstrate through several interesting and realistic examples that the developed type theory can be effectively put into practice. In Section 6, we discuss how the code reentrancy problem is addressed in the presence of concurrency. Lastly, we mention some related work and conclude. In addition, we list the complete typing rules and more code examples in the appendix to facilitate assessment.

types	T	$::=$	$\delta^i \mid T_1 * T_2 \mid VT_1 \rightarrow_i VT_2 \mid$
viewtypes	VT	$::=$	$\delta^l \mid T \mid VT_1 \otimes VT_2 \mid VT_1 \rightarrow_l VT_2$
expressions	e	$::=$	$c(e_1, \ldots, e_n) \mid r \mid xf \mid \mathbf{if}(e_1, e_2, e_3) \mid \mathbf{fst}(e) \mid$ $\mathbf{snd}(e) \mid \langle e_1, e_2 \rangle \mid \mathbf{lam}\, x.\, e \mid e_1(e_2) \mid \mathbf{fix}\, f.\, v \mid$ $\mathbf{let}\ \langle x_1, x_2 \rangle = e_1\ \mathbf{in}\ e_2\ \mathbf{end}$
values	v	$::=$	$cc(\overline{v}) \mid r \mid x \mid \langle v_1, v_2 \rangle \mid \mathbf{lam}\, x.\, e$
intuitionistic. exp. ctx.	Γ	$::=$	$\emptyset \mid \Gamma, xf : T$
linear. exp. ctx.	Δ	$::=$	$\emptyset \mid \Delta, x : VT$

Fig. 1. The syntax of $\mathcal{L}_0$

2 The Starting Point: $\mathcal{L}_0$

We first present a language $\mathcal{L}_0$ with a simple linear type system, using it as a starting point to set up some machinery for further development. We do not address the issue of resource sharing in $\mathcal{L}_0$, which is to be done at the next stage. The syntax of the language $\mathcal{L}_0$ is given in Figure 1.

We use c for constants, which include both constant functions cf and constant constructors cc, and r for resources. Note that we treat the resources in $\mathcal{L}_0$ abstractly. For instance, when dealing with memory manipulation at low level, we introduce resources of the form $v@L$, where v and L range over values and memory addresses (represented as natural numbers), respectively. Intuitively, $v@L$ means that the value v is stored at the address L. For a simple and clean presentation, we assume in this paper that values are properly boxed and can thus be stored in one memory unit. In practice, we can and do handle unboxed values without much complication.

We use T and VT for types and viewtypes, respectively, and δ^i and δ^l for base types and base viewtypes respectively, where the superscript i means intuitionistic while l means linear. For instance, **bool** and **int** are base types for booleans and integers while **int**$@L$ is a base viewtype for the resource $v@L$ given v is of type **int**.

Note that a type is always considered a viewtype. At this point, we emphasize that $\rightarrow_l$ should not be confused with the linear implication $\multimap$ in linear logic. Given $VT_1 \rightarrow_l VT_2$, the viewtype constructor $\rightarrow_l$ simply indicates that $VT_1 \rightarrow_l VT_2$ itself is a viewtype. The meaning of various forms of types and viewtypes is to be made clear and precise when the rules are presented for assigning viewtypes to expressions in $\mathcal{L}_0$.

We assume the existence of a signature SIG that assigns each resource r a base viewtype δ^l and each constant c a constant type (c-type, for short) of the form $(VT_1, \ldots, VT_n) \Rightarrow VT$ (VT must be either δ^i or δ^l if c is a constructor), where n is the arity of c. For instance, the truth values *true* and *false* are assigned the c-type $() \Rightarrow$ **bool**.

The expressions e and values v in $\mathcal{L}_0$ are mostly standard. We use x for lam-variables and f for fix-variables, where the former is a value but the latter is not, and write xf for either x or f. We may write $\overline{v}$ for a (possibly empty) sequence of values.

We use R for finite *multisets* of resources. Given R_1 and R_2, we write $R_1 \uplus R_2$ for the *multiset union* of R_1 and R_2. Given an expression e, we use $\rho(e)$ for the multiset of resources contained in e, which is defined as follows:

$$\begin{array}{rclrcl}
\rho(c(e_1,\ldots,e_n)) & = & \rho(e_1) \uplus \ldots \uplus \rho(e_n) & \rho(r) & = & \{r\} \\
\rho(x) & = & \emptyset & \rho(\mathbf{if}(e_1,e_2,e_3)) & = & \rho(e_1) \uplus \rho(e_2) \\
\rho(\mathbf{let}\ \langle x_1,x_2\rangle = e_1\ \mathbf{in}\ e_2\ \mathbf{end}) & = & \rho(e_1) \uplus \rho(e_2) & \rho(\mathbf{lam}\, x.\, e) & = & \rho(e) \\
\rho(e_1(e_2)) & = & \rho(e_1) \uplus \rho(e_2) & \ldots & = & \ldots
\end{array}$$

In the case where $e = \mathbf{if}(e_1,e_2,e_3)$, the type system of $\mathcal{L}_0$ is to enforce that $\rho(e_2) = \rho(e_3)$ if e can be assigned a viewtype, and this is the reason for defining $\rho(\mathbf{if}(e_1,e_2,e_3))$ as $\rho(e_1) \uplus \rho(e_2)$.

We emphasize that resources are *not* necessarily preserved under evaluation. It is possible for an expression containing resources to be assigned a type or an expression containing no resources to be assigned a viewtype. For instance, suppose that *alloc* is a constant function that takes a natural number as its argument and returns some resources. Then the expression *alloc*(1) contains no resources but it cannot be assigned a type (as the evaluation of *alloc*(1) returns a value containing a resource $v@L$ for some value v and address L).

It is clear that we cannot combine resources arbitrarily. For instance, it is impossible to have resources $v_1@L$ and $v_2@L$, simultaneously. We define **ST** as a collection of finite multisets of resources and assume that $\emptyset \in \mathbf{ST}$ and **ST** is closed under subset relation, that is, for any R_1 and R_2, $R_2 \in \mathbf{ST}$ if $R_1 \in \mathbf{ST}$ and $R_2 \subseteq R_1$, where $\subseteq$ is the subset relation on multisets. We say that R is a valid multiset of resources if $R \in \mathbf{ST}$ holds. Note that the definition of **ST** is considered abstract, which is not specific to the language. For instance, if the type system is used for reasoning about memory manipulation, **ST** can be defined as the collection of all valid memory states.

Dynamic Semantics. The definition of evaluation contexts E in $\mathcal{L}_0$ is given as follows:

$$\begin{array}{l}
E ::= [] \mid cc(v_1,\ldots,v_{i-1},E,e_{i+1},\ldots,e_n) \mid \mathbf{if}(E,e_1,e_2) \mid \langle E,e\rangle \mid \langle v,E\rangle \mid \\
\quad \mathbf{fst}(E) \mid \mathbf{snd}(E) \mid \mathbf{let}\ \langle x_1,x_2\rangle = E\ \mathbf{in}\ e\ \mathbf{end} \mid E(e) \mid v(E)
\end{array}$$

We are to use evaluation contexts to define the (call-by-value) dynamic semantics of $\mathcal{L}_0$. There are two forms of redexes in $\mathcal{L}_0$: pure redexes and *ad hoc* redexes. The pure redexes and their reducts are defined as follows:

- $\mathbf{if}(\mathit{true},e_1,e_2)$ is a pure redex, and its reduct is e_1.
- $\mathbf{if}(\mathit{false},e_1,e_2)$ is a pure redex, and its reduct is e_2.
- $\mathbf{let}\ \langle x_1,x_2\rangle = \langle v_1,v_2\rangle\ \mathbf{in}\ e\ \mathbf{end}$ is a pure redex, and its reduct is $e[x_1,x_2 \mapsto v_1,v_2]$.
- $\mathbf{fst}(\langle v_1,v_2\rangle)$ is a pure redex, and its reduct is v_1.
- $\mathbf{snd}(\langle v_1,v_2\rangle)$ is a pure redex, and its reduct is v_2.
- $(\mathbf{lam}\, x.\, e)(v)$ is a pure redex, and its reduct is $e[x \mapsto v]$.
- $\mathbf{fix}\, f.\, v$ is a pure redex, and its reduct is $v[f \mapsto \mathbf{fix}\, f.\, v]$.

Evaluating calls to constant functions is of particular importance in $\mathcal{L}_0$ as it may involve resource generation and consumption. Assume that cf is a constant function of arity n. The expression $cf(v_1,\ldots,v_n)$ is an ad hoc redex if *cf* is defined at $v_1, \ldots, v_n$, and any value v of $cf(v_1, \ldots, v_n)$ is a reduct of $cf(v_1, \ldots, v_n)$. For instance, *alloc* is a function that takes a natural number n to return a pointer to some address L associated with a tuple of resources $\langle v_0@L, v_1@L+1, \ldots, v_{n-1}@L+n-1\rangle$ for some values $v_0, v_1, \ldots, v_{n-1}$, that is, $alloc(n)$ reduces to a pointer that points to n consecutive memory units containing some unspecified values.

Definition 1. *Given expressions* e_1 *and* e_2*, we write* $e_1 \hookrightarrow e_2$ *if* $\rho(e_1) \in \mathbf{ST}$ *holds,* $e_1 = E[e]$ *and* $e_2 = E[e']$ *for some evaluation context* E *and expressions* e, e' *such that* e *is a redex,* e' *is a reduct of* e *and* $\rho(E[e']) \in \mathbf{ST}$ *holds.*

As usual, we use $\hookrightarrow^*$ for the reflexive and transitive closure of $\hookrightarrow$. We say that e reduces to e' purely if the redex being reduced is pure. A type system is to be developed to guarantee that resources are preserved under pure reduction, that is, $\rho(e) = \rho(e')$ whenever e reduces to e' purely. However, resources may be generated as well as consumed when ad hoc reduction occurs. Suppose that $e_1 = E[alloc(1)]$ and $v@L$ occurs in E. Though $\langle v@L, L\rangle$ is a reduct of *alloc*, we cannot allow $e_1 \hookrightarrow E[\langle v@L, L\rangle]$ as the resource $v@L$ occurs repeatedly in $E[\langle v@L, L\rangle]$. This is precisely the reason that we require $\rho(e_2) \in \mathbf{ST}$ whenever $e_1 \hookrightarrow e_2$ holds.

Static Semantics. An intuitionistic expression context Γ can be treated as a finite mapping that maps xf to T for each declaration $xf : T$ in Γ, and we use $\mathbf{dom}(\Gamma)$ for the domain of Γ. A linear expression context Δ can be treated in the same manner. Given an intuitionistic expression context Γ and a linear expression context Δ such that $\mathbf{dom}(\Gamma) \cap \mathbf{dom}(\Delta) = \emptyset$, we can form an expression context $(\Gamma; \Delta)$. Clearly, we can also treat expression contexts as finite mappings. Given Γ and Δ, we use $(\Gamma; \Delta), x : VT$ for either $(\Gamma; \Delta, x : VT)$ or $(\Gamma, x : VT; \Delta)$ (if VT is actually a type).

We use $\Gamma; \Delta \vdash e : VT$ for a judgment stating that the viewtype VT can be assigned to e under $(\Gamma; \Delta)$. The rules for assigning viewtypes to expressions in $\mathcal{L}_0$ are largely standard thus omitted. We provide some explanation for the following two:

$$\frac{\Gamma; \Delta, x : VT_1 \vdash e : VT_2}{\Gamma; \Delta \vdash \mathbf{lam}\, x.\, e : VT_1 \rightarrow_l VT_2}\ \textbf{(ty-}\rightarrow_l\textbf{-intr)} \qquad \frac{\Gamma; \emptyset, x : VT_1 \vdash e : VT_2 \quad \rho(e) = \emptyset}{\Gamma; \emptyset \vdash \mathbf{lam}\, x.\, e : VT_1 \rightarrow_i VT_2}\ \textbf{(ty-}\rightarrow_i\textbf{-intr)}$$

Given two viewtypes VT_1 and VT_2, $VT_1 \rightarrow_l VT_2$ is a viewtype and $VT_1 \rightarrow_i VT_2$ is a type. The rules **(ty-**$\rightarrow_l$**-intr)** and **(ty-**$\rightarrow_i$**-intr)** assign a viewtype and type to a function, respectively; the function can use its argument many times if VT_1 is a type or exactly once if VT_1 is viewtype. Intuitively, when the rule **(ty-**$\rightarrow_i$**-intr)** is applied, the body of the involved function must contain no resources as the function is a value to which a type (not just a viewtype) is assigned.

Soundness. As usual, the soundness of the type system of $\mathcal{L}_0$ rests on the following two theorems. The detailed proof can be found in [13].

Theorem 1 (Subject Reduction). *Assume that* $\emptyset; \emptyset \vdash e : VT$ *is derivable,* $\rho(e) \in \mathbf{ST}$ *and* $e \hookrightarrow e'$*. Then* $\emptyset; \emptyset \vdash e' : VT$ *is also derivable and* $\rho(e') \in \mathbf{ST}$.

Theorem 2 (Progress). *Assume that $\emptyset; \emptyset \vdash e : VT$ is derivable and $\rho(e) \in \mathbf{ST}$. Then either e is a value or $e \hookrightarrow e'$ holds for some expression e'.*

3 Supporting Resource Sharing: $\mathcal{L}_\Box$

The need for resource sharing occurs immediately in practice. As mentioned in Section [1], we must employ some form of resource sharing when implementing sharable data structures. We introduce a form of modality $\Box$, which we refer to as *sharing modality*, to support resource sharing. We first give some intuitive but rather informal explanation about the expected use of $\Box$. Given a value v of viewtype VT, we can imagine that a box is created to store the value v. We use h for the handle of the box, which is assigned the type $\Box VT$ and can thus be duplicated. The unary type constructor $\Box$, which takes a viewtype to form a type, imposes the following requirement on a program that attempts to access the value stored in the box through the handle h of a box: the program can take out the value in the box and manipulate it freely as long as it guarantees to return to the box a (possibly different) value of the same viewtype at the end of its evaluation.

With the sharing modality, there is an interesting but troubling problem of *double borrow* that must be properly addressed. Suppose that we are at a point where the value stored in a box has already been borrowed out but no value has been returned to the box yet. At this point, if there is another request to borrow from the box, then a scenario of double borrow occurs, which makes the following misbehaved program possible:

```
let □r₁ = x in   (* let □ is the syntax to borrow resource from a boxed value *)
  let □r₂ = x in
    ... free(r₁) ... in ... access(r₂) end end end
```

where the resource boxed in x is double borrowed and bound to r_1 and r_2, thus, both referring exactly the same resource. Furthermore, the program subsequently frees r_1 before accessing r_2, which is clearly a safety violation.

In order to establish the soundness of a type system accommodating the sharing modality, we must prevent double borrow from ever happening. We achieve this by employing a notion of *types with effects* [9]. Specifically, we decorate the viewtype constructor $\rightarrow_l$ with a bit b ranging over 0 and 1. Given a function of viewtype $VT_1 \xrightarrow{b}_l VT_2$, the evaluation of a call to this function is guaranteed to borrow no values from any boxes if $b = 0$. Otherwise, it may borrow values from some boxes. We decorate the type constructor $\rightarrow_i$ with a bit in precisely the same manner.

The language $\mathcal{L}_0$ is extended to $\mathcal{L}_\Box$ with some additional syntax in Figure [2]. We use h for handles (of boxes) and assume that there exist infinitely many of them that can be generated freshly. Given an expression **let** $\Box x = e_1$ **in** e_2 **end**, we expect that e_1 evaluates to a handle h, and then x is bound to the value stored in the box with the handle h and e_2 evaluates to a pair $\langle v_1, v_2 \rangle$, and then v_1 is inserted into the box and v_2 is the value of the expression **let** $\Box x = e_1$ **in** e_2 **end**.

types	$T ::= \ldots \mid \Box VT \mid VT_1 \xrightarrow{b}_l VT_2 \mid VT_1 \xrightarrow{b}_i VT_2$
expressions	$e ::= \ldots \mid h \mid \Box e \mid \textbf{let}\ \Box x = e_1\ \textbf{in}\ e_2\ \textbf{end}$
values	$v ::= \ldots \mid h$
eval. ctx.	$E ::= \ldots \mid \Box E \mid \textbf{let}\ \Box x = E\ \textbf{in}\ e\ \textbf{end}$

Fig. 2. The additional syntax for $\mathcal{L}_\Box$

We are to use the type system of $\mathcal{L}_\Box$ to guarantee that the evaluation of e_2 does not borrow values from any boxes.

We use $\mathcal{M}$ for stores, which are finite mappings from handles to values or a special symbol $\bullet$. Given a store $\mathcal{M}$ and a handle h in the domain $\mathbf{dom}(\mathcal{M})$ of $\mathcal{M}$, we say that $\mathcal{M}$ is *available* at h if and only if $\mathcal{M}(h) = v$ for some value v. We say that $\mathcal{M}$ is full if $\mathcal{M}$ is available at each $h \in \mathbf{dom}(\mathcal{M})$. In the following presentation, we only deal with stores that are either full or not available at only one handle. We use $\mathcal{M}[h \mapsto v]$ for the mapping that extends $\mathcal{M}$ with an extra link from h to v, where $h \notin \mathbf{dom}(\mathcal{M})$ is assumed. In addition, we use $\mathcal{M}[h := v^*]$ for the mapping $\mathcal{M}'$ such that $\mathcal{M}'(h) = v^*$ and $\mathcal{M}'(h') = \mathcal{M}(h')$ for each $h' \in \mathbf{dom}(\mathcal{M}) = \mathbf{dom}(\mathcal{M}')$ that is not h, where v^* ranges over values and the special symbol $\bullet$. We say that $\mathcal{M}'$ extends $\mathcal{M}$ if $\mathcal{M}'(h) = \mathcal{M}(h)$ for each $h \in \mathbf{dom}(\mathcal{M}) \subseteq \mathbf{dom}(\mathcal{M}')$.

We extend the definition of ρ to deal with the new syntax: $\rho(h) = \emptyset$, $\rho(\Box e) = \rho(e)$, $\rho(\textbf{let}\ \Box x = e_1\ \textbf{in}\ e_2\ \textbf{end}) = \rho(e_1) \uplus \rho(e_2)$, $\rho(\bullet) = \emptyset$ and $\rho(\mathcal{M}) = \uplus_{h \in \mathbf{dom}(\mathcal{M})} \rho(\mathcal{M}(h))$.

We use $\hat{e}$ for *intermediate expressions*, which are either closed expressions e or triples of the form $\langle E, h, e \rangle$, where E, h, e range over evaluation contexts, handles and closed expressions, respectively. We define $\rho(\hat{e})$ to be $\rho(e)$ if $\hat{e} = e$ or $\rho(E[e])$ if $\hat{e} = (E, h, e)$. In $\mathcal{L}_\Box$, the evaluation relation $\hookrightarrow$ is defined on pairs of the form $\langle \mathcal{M}, \hat{e} \rangle$.

We say that $\mathcal{M}$ matches $\hat{e}$ if either $\hat{e} = e$ for some e and $\mathcal{M}$ is full or $\hat{e} = \langle E, h, e \rangle$ for some E, h, e and $\mathcal{M}$ is not available only at h.

Definition 2. *(Reduction in $\mathcal{L}_\Box$) We say that $(\mathcal{M}, \hat{e})$ reduces to $(\mathcal{M}', \hat{e}')$ if $(\mathcal{M}, \hat{e}) \hookrightarrow (\mathcal{M}, \hat{e}')$, which is defined as follows:*

- *If e reduces to e' and $\rho(\mathcal{M}) \uplus \rho(e') \in \mathbf{ST}$, then $(\mathcal{M}, e) \hookrightarrow (\mathcal{M}, e')$.*
- *If e reduces to e' and $\rho(\mathcal{M}) \uplus \rho(E[e']) \in \mathbf{ST}$, then $(\mathcal{M}, \langle E, h, e \rangle) \hookrightarrow (\mathcal{M}, \langle E, h, e' \rangle)$.*
- *If $h \notin \mathbf{dom}(\mathcal{M})$, then $(\mathcal{M}, E[\Box v]) \hookrightarrow (\mathcal{M}[h \mapsto v], E[h])$.*
- *If $\mathcal{M}(h) = v$, then $(\mathcal{M}, E[\textbf{let}\ \Box x = h\ \textbf{in}\ e\ \textbf{end}]) \hookrightarrow (\mathcal{M}[h := \bullet], \langle E, h, e[x \mapsto v] \rangle)$.*
- *If $\mathcal{M}(h) = \bullet$, then $(\mathcal{M}, \langle E, h, \langle v_1, v_2 \rangle \rangle) \hookrightarrow (\mathcal{M}[h := v_1], E[v_2])$.*

It is clear that an intermediate expression of the form $\langle E, h, e[x \mapsto v] \rangle$ is generated when we evaluate $(M, E[\textbf{let}\ \Box x = h\ \textbf{in}\ e\ \textbf{end}])$, where v is the value stored in the box with the handle h; we are disallowed to borrow values from any boxes when evaluating $e[x \mapsto v]$; the expression $e[x \mapsto v]$ is expected to evaluate to a pair $\langle v_1, v_2 \rangle$, allowing v_1 to be inserted into the box with the handle h and the evaluation of $(M[h := v_1], E[v_2])$ to start.

It may be argued that the restriction is too severe that disallows borrowing values from any boxes during the evaluation of $\langle E, h, e[x \mapsto v]\rangle$ as it clearly suffices to only disallow borrowing values from the box with the handle h. However, it is highly nontrivial to use the type of an expression to indicate from which boxes values can or cannot be borrowed during the evaluation of the expression. Also, it is unclear whether there are strong practical reasons to do so. For instance, we can always extract values from two shared resources in sequence (and manipulate the values thereafter) instead of accessing them at the same time. In essence, the shared resources constructed through the modality can be used exactly the same way as ones available in ML such as references and arrays.

Static Semantics. We use μ for store types, which are finite mappings from handles to viewtypes. Also, we write $\mu[h \mapsto VT]$ for the mapping that extends μ with an extra link from h to VT, where $h \notin \mathbf{dom}(\mu)$ is assumed. We say that μ' extends μ if $\mu'(h) = \mu(h)$ for each $h \in \mathbf{dom}(\mu) \subseteq \mathbf{dom}(\mu')$.

We use $\Gamma; \Delta \vdash^b_\mu e : VT$ for judgments assigning viewtypes to expressions, where the bit b ranges over 0 and 1. Given two bits b_1 and b_2, the bit $b_1 \oplus b_2$ is 0 if and only if $b_1 = b_2 = 0$. In other words, $\oplus$ is the `OR` function on bits. The rule of most interest is

$$\frac{\Gamma; \Delta_1 \vdash^b_\mu e_1 : \Box VT_1 \quad \Gamma; \Delta_2, x : VT_1 \vdash^0_\mu e_2 : VT_1 \otimes VT_2}{\Gamma; \Delta_1 \uplus \Delta_2 \vdash^1_\mu \mathbf{let}\ \Box x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} : VT_2}\ \textbf{(ty-}\Box\textbf{-elim)}$$

where the second premise states that e_2 must be *borrow-free* (with the bit 0) while the final judgement is with bit 1, indicating some resource is borrowed. The complete rules for assigning viewtypes to expressions in $\mathcal{L}_\Box$ are omitted for brevity. We write $\vdash \mathcal{M} : \mu$ to mean that for each $h \in \mathbf{dom}(\mathcal{M}) = \mathbf{dom}(\mu)$, either $\mathcal{M}(h) = \bullet$ or $\emptyset; \emptyset \vdash^0_\mu \mathcal{M}(h) : \mu(h)$ is derivable.

Soundness. As usual, the soundness of $\mathcal{L}_\Box$ rests on the following theorems. The detailed proof can be found in [13].

Theorem 3 (Subject Reduction). *Assume that $\vdash \mathcal{M} : \mu$ holds and $\vdash^b_\mu \hat{e} : VT$ is derivable. If $(\mathcal{M}, \hat{e}) \hookrightarrow (\mathcal{M}', \hat{e}')$, then there exists μ' extending μ such that $\vdash \mathcal{M}' : \mu'$, $\rho(\mathcal{M}') \uplus \rho(\hat{e}') \in \mathbf{ST}$ and $\vdash^{b'}_{\mu'} \hat{e}' : VT$ is derivable for some $b' \leq b$.*

Theorem 4 (Progress). *Assume that $\vdash \mathcal{M} : \mu$ holds, $\mathcal{M}$ matches $\hat{e}$, $\rho(\mathcal{M}) \uplus \rho(\hat{e}) \in \mathbf{ST}$ and $\vdash^b_\mu \hat{e} : VT$ is derivable. Then $(\mathcal{M}, \hat{e}) \hookrightarrow (\mathcal{M}', \hat{e}')$ for some $\mathcal{M}'$ and $\hat{e}'$.*

4 Extensions

While the basic design for supporting safe resource sharing in programming is already demonstrated in $\mathcal{L}_\Box$, it is nonetheless difficult to truly reap the benefits of this design given that the type system of $\mathcal{L}_\Box$ is simply too limited. We need two extensions when presenting some interesting and realistic examples in Section 5.

We first extend $\mathcal{L}_\Box$ to $\mathcal{L}^{\forall,\exists}_\Box$ by incorporating universally as well as existentially quantified viewtypes, which include both polymorphic viewtypes and dependent

sorts	$\sigma ::= \mathit{bool} \mid \mathit{int} \mid \mathit{addr} \mid \mathit{type} \mid \mathit{viewtype}$
types	$T ::= \ldots \mid a \mid B \supset T \mid \forall a : \sigma.\ T \mid B \wedge T \mid \exists a : \sigma.\ T$
viewtypes	$VT ::= \ldots \mid a \mid B \supset VT \mid \forall a : \sigma.\ VT \mid B \wedge VT \mid \exists a : \sigma.\ VT$
expressions	$e ::= \ldots \mid \supset^+(v) \mid \supset^-(e) \mid \forall^+(v) \mid \forall^-(e) \mid$ $\wedge(e) \mid \textbf{let}\ \wedge(x) = e_1\ \textbf{in}\ e_2\ \textbf{end} \mid \exists(e) \mid \textbf{let}\ \exists(x) = e_1\ \textbf{in}\ e_2\ \textbf{end}$
values	$v ::= \ldots \mid \supset^+(v) \mid \forall^+ v \mid \wedge(v) \mid \exists(v)$

Fig. 3. The additional syntax of $\mathcal{L}_{\Box}^{\forall,\exists}$

viewtypes. The extra syntax of $\mathcal{L}_{\Box}^{\forall,\exists}$ (over that of $\mathcal{L}_{\Box}$) is given in Figure 3. Following the work on the framework Applied Type System ($\mathcal{ATS}$) [19,20], this extension is largely standard.

In order to effectively deal with memory manipulation at low level, we also need to support a paradigm that combines programming with theorem proving. We introduce a language $\mathcal{L}_{\Box,pf}^{\forall,\exists}$ (detailed syntax are omitted for brevity) by extending $\mathcal{L}_{\Box}^{\forall,\exists}$ with a component in which only pure and total programs can be constructed, and this component is referred to as the theorem-proving component of $\mathcal{L}_{\Box,pf}^{\forall,\exists}$. Proofs (pure and total dynamic terms) can be constructed in the theorem-proving component to attest various properties of programs (effectful and nonterminating dynamic terms). Due to the space constraint, we cannot give a detailed account of this paradigm in this paper. Instead, we refer the interested reader to [3] for theoretical details and practical examples. Note that the proofs in $\mathcal{L}_{\Box,pf}^{\forall,\exists}$ are only needed for type-checking purpose and they are completely erased before run-time execution.

5 Examples

In this section, we present several examples taken from the programming language ATS [20] to give the reader some concrete feel as to how the developed type theory for resource sharing can be put into practice. All examples are presented in the concrete syntacx of ATS, which is inspired by the syntax of Standard ML [11]. A (partial) type inference process [2] is used to elaborate programs written in the concrete syntax into the formal syntax of $\mathcal{L}_{\Box,pf}^{\forall,\exists}$.

We assume the existence of primitive memory access functions `ptr_get` and `ptr_set` of the following types:

$$\begin{array}{lcl} \texttt{ptr\_get} & : & \forall\tau.\forall\lambda.\ (\tau@\lambda, \mathbf{ptr}(\lambda)) \Rightarrow (\tau@\lambda)\otimes\tau \\ \texttt{ptr\_set} & : & \forall\tau.\forall\lambda.\ (\mathbf{top}@\lambda, \mathbf{ptr}(\lambda), \tau) \Rightarrow (\tau@\lambda)\otimes\mathbf{1} \end{array}$$

where we use τ and λ for bounded static variables of sorts *type* and *addr*, respectively, and **top** for the type such that every type is a subtype of **top**. Basically, `ptr_get` reads through a pointer and `ptr_set` writes through a pointer. Given T and L, `ptr_get` takes a proof of $T@L$ and a pointer to L and it returns a proof of $T@L$ and a value of type T. So a linear proof of $T@L$ threads through

```
fun ref_make {a:type} (x: a): ref a = let
   val (pf | p) = alloc (1) // pf: array_v (top, 1, l) and p: ptr l
   // The pattern matching is verified to be exhaustive
   prval ArraySome (pf1, ArrayNone ()) = pf // pf1: top @ l
   val (pf1 | ()) = ptr_set (pf1 | p, x) // pf1: a @ l
in (vbox pf1 | p) end

fun ref_get {a:type} (r: ref a): a = let
   val (h | p) = r // h: vbox(a @ l) for some l
   prval vbox pf = h // (pf : a @l) is borrowed
in ptr_get (pf | p) end

fun ref_set {a:type} (r: ref a, x: a): void = let
   val (h | p) = r // h: vbox(a @ l) for some l
   prval vbox pf = h // (pf: a @ l) is borrowed
in ptr_set {a} (pf | p, x) end
```

Fig. 4. An implementation of references

`ptr_get`. On the other hand, given T and L, `ptr_set` takes a proof of $\mathbf{top}@L$, a value of type T and a pointer to L and it returns a proof of $T@L$ and the unit. So a proof of $\mathbf{top}@L$ is consumed and then a proof of $T@L$ is generated by `ptr_set`.

In order to model more sophisticated memory layouts, we need to form recursive views. For instance, we may declare a (dependent) view constructor *array_v*: Given a type T, an integer I and an address L, $\mathit{array\_v}(T, I, L)$ forms a view stating that there are I values of type T stored at consecutive addresses $L, L+1, \ldots, L+I-1$. There are two proof constructors *ArrayNone* and *ArraySome* associated with *array_v*, which are formally assigned the following c-types:

$$\begin{array}{l} \mathit{ArrayNone} : \forall\lambda.\forall\tau.() \Rightarrow \mathit{array\_v}(\tau, 0, \lambda) \\ \mathit{ArraySome} : \forall\lambda.\forall\tau.\forall\iota.\iota \geq 0 \supset (\tau@\lambda, \mathit{array\_v}(\tau, \iota, \lambda+1)) \Rightarrow \mathit{array\_v}(\tau, \iota+1, \lambda) \end{array}$$

Intuitively, *ArrayNone* is a proof of $\mathit{array\_v}(T, 0, L)$ for any type T and address L, and $\mathit{ArraySome}(v_1, v_2)$ is a proof of $\mathit{array\_v}(T, I+1, L)$ for any type T, integer I and address L if v_1 and v_2 are proofs of views $T@L$ and $\mathit{array\_v}(T, I, L+1)$, respectively.

References. In Figure 4, we give an implementation of references (as commonly supported in ML) in ATS. The symbol | in the code is a separator (just like a comma), which separates proofs from values (so as to make the code easier to read).

Given a type T, we use $\mathbf{ref}(T)$ for the type of references to values of type T, which is formally defined as $\exists\lambda.\Box(T@\lambda) * \mathbf{ptr}(\lambda)$. Therefore, a reference is just a pointer to some address L paired with the handle of a box containing a proof stating that a value of type T is stored at L. The three implemented functions in Figure 4 are given the following types, respectively:

$$\mathit{ref\_make} : \forall\tau.\ \tau \xrightarrow{0}_i \mathbf{ref}(\tau) \quad \mathit{ref\_get} : \forall\tau.\ \mathbf{ref}(\tau) \xrightarrow{1}_i \tau \quad \mathit{ref\_set} : \forall\tau.\ \mathbf{ref}(\tau) * \tau \xrightarrow{1}_i \mathbf{1}$$

We use the keyword `prval` to introduce bindings on proof variables. As such bindings are to be erased before run-time, ATS automatically verifies that the pattern matching involved is exhaustive.

Clearly, *ref_get* and *ref_set* are operationally equivalent (after types and proofs are erased) to `ptr_get` and `ptr_set`, respectively, which is expected. What we find a bit surprising is that a feature as simple and common as references can involve so much type theory (on universal and existential types, linear types, sharing modality, programming with theorem proving, etc.).

Sharable Arrays. We give an implementation of sharable arrays in ATS. The following two functions:

$$\begin{aligned}\texttt{array\_init} &: \forall\tau.\forall\iota.\forall\lambda.\ \iota \geq 0 \supset (\mathit{array\_v}(\mathbf{top}, \iota, \lambda) * \mathbf{ptr}(\lambda) * \tau \xrightarrow{0}_i \mathit{array\_v}(\tau, \iota, \lambda) * \mathbf{1})\\ \texttt{array\_get} &: \forall\tau.\forall\iota_1.\forall\iota_2.\forall\lambda.\ 0 \leq \iota_2 \wedge \iota_2 < \iota_1 \supset\\ &\qquad (\mathit{array\_v}(\tau, \iota_1, \lambda) * \mathbf{ptr}(\lambda) * \mathbf{int}(\iota_2) \xrightarrow{0}_i \mathit{array\_v}(\tau, \iota_1, \lambda) * \tau)\end{aligned}$$

are needed in the implementation. However, the code for implementing them is omitted here for brevity.

The sharable array can be implemented similarly. Given a type T, we define $\mathbf{SArray}(T)$ as

$$\exists\lambda.\exists\iota.\ \iota \geq 0 \wedge (\Box(\mathbf{int}(\iota)@(\lambda - 1) \otimes \mathit{array\_v}(T, \iota, \lambda)) * \mathbf{ptr}(\lambda))$$

That is, we represent a sharable array as a pointer to some address L associated with a box of two proofs showing that a natural number I is stored at $L-1$ and an array of size I starts at L, where $v_1, \ldots, v_I$ are values stored in the array. The following functions are implemented for array creation and subscription:

$$\texttt{SArray\_make} \ :\ \forall\tau.\ \mathbf{Nat} * \tau \xrightarrow{0}_i \mathbf{SArray}(\tau) \qquad \texttt{SArray\_get} \ :\ \forall\tau.\ \mathbf{SArray}(\tau) * \mathbf{Int} \xrightarrow{1}_i \tau$$

where **Nat** and **Int** are defined as $\exists\iota.\ \iota \geq 0 \wedge \mathbf{int}(\iota)$ and $\exists\iota.\ \mathbf{int}(\iota)$, respectively. For brevity, the actual implementation is omitted.

Similarly, back to our motivating examples in Section 1, the *malloc* and *free* functions can now be given the ideal types (no free list threaded through) with the help of the sharing modality. The free list, as a linear resource, will be internal to *malloc* and *free* in which complex memory manipulations can be reasoned and ensured to be safe by linear types. While, the free list is completely hidden from the client point of view so that both *malloc* and *free* can be readily used in practical programming.

6 Code Reentrancy

While we have so far only studied the sharing modality $\Box$ in sequential programming, a major intended use of the modality is actually in multi-threaded programming. We attempt to give an intuitive but rather informal explanation on this issue as a formalized account for multi-threaded programming is simply beyond the scope of the paper. We refer the interested readers to [14] for a theoretical development and practical examples of multi-threaded programming in ATS.

As mentioned earlier, the sharing modality $\Box$ is unable to support safe resource sharing in concurrent programming. The main reason is that a thread evaluating an expression of the form **let** $\Box x = h$ **in** e **end** may be suspended at a time when the box with the handle h is empty, and meanwhile another thread may attempt to borrow from the box and thus result in a case of double borrow. In other words, code of the form **let** $\Box x = h$ **in** e **end** is in general unsafe to be executed in a thread. However, if e can be evaluated atomically, that is, without the possibility of suspension during the evaluation of e, then it is safe to execute code of the form **let** $\Box x = h$ **in** e **end** as the problem of double borrow can no longer occur. We give some common cases where this can happen.

– If e can be compiled into a single atomic instruction, then the evaluation of e is guaranteed to be atomic. For instance, in the example on implementing references, the body of *ref_get* (*ref_set*) may be compiled into a single atomic read (write) instruction on memory. If this is true, then *ref_get* (*ref_set*) can be safely used in threads.
– Some hardware support (e.g., disabling interrupts) can be employed to ensure that the evaluation of e is atomic. For instance, the function *kmalloc* in Linux (for allocating memory in kernel space) is often implemented to be reentrant by disabling interrupts (on a single core machine) during its execution.
– A thread may use signals to put all other threads into sleep before executing non-reentrant code and then wakes them up after the execution. For instance, a thread that does garbage collection often makes use of this strategy.

Let us now introduce a half bit .5 for decorating typing judgments. Note that $b_1 \oplus b_2$ is defined as $max(b_1, b_2)$. In addition to the previously presented typing rules, we also add the following one

$$\frac{\begin{array}{c}\Sigma; \overline{B}; (\Gamma; \Delta_1) \vdash^{b}_{\mu} e_1 : \Box VT_1 \\ \Sigma; \overline{B}; (\Gamma_2; \Delta_2, x : VT_1) \vdash^{0}_{\mu} e_2 : VT_1 \otimes VT_2\end{array}}{\Sigma; \overline{B}; (\Gamma; \Delta_1 \uplus \Delta_2) \vdash^{.5 \oplus b}_{\mu} \textbf{let } \Box x = e_1 \textbf{ in atm}(e_2) \textbf{ end} : VT_2} \quad \textbf{(ty-}\Box\textbf{-elim-atom)}$$

where $\mathbf{atm}(e)$ indicates that e is an expression that must be evaluated atomically (possibly with some unspecified hardware/software support). We name this rule **(ty-$\Box$-elim-atom)**. Suppose that $\emptyset; \emptyset; (\emptyset; \emptyset) \vdash^{b}_{\mu} e$ is derivable. Then we expect e to be reentrant if $b < 1$ and non-reentrant if $b = 1$. So for a function of type $VT_1 \xrightarrow{b}_l VT_2$ or $VT_1 \xrightarrow{b}_i VT_2$, the function is reentrant if $b < 1$ and non-reentrant if $b = 1$. If thread creation only accepts functions of the viewtype $\mathbf{1} \xrightarrow{b}_l \mathbf{1}$ for some $b < 1$, then non-reentrant functions are prevented from being called within threads.

7 Related Work and Conclusion

A fundamental issue in programming is on program verification, that is, verifying (in an effective manner) whether a program meets its specification. In general, existing approaches to program verification can be classified into two categories. In one category, the underlying theme is to develop a proof theory based on

Hoare logic (or its invariants) for reasoning about imperative stateful programs. In the other category, the focus is on developing a type theory that allows the use of types in capturing program properties.

In [22], we outlined a type system *ATS/SV*, which is rather similar to $\mathcal{L}^{\forall,\exists}_{\Box,pf}$ minus the sharing modality. *ATS/SV* is effective in supporting memory manipulation through pointers, and a variety of mutable data structures (e.g., linked lists, splay trees, etc.) are implemented in *ATS/SV*. However, resource sharing cannot be properly dealt with in *ATS/SV*, causing serious difficulties in practice. The sharing modality $\Box$ is introduced precisely for the purpose of addressing this limitation in *ATS/SV*.

The sharing modality in our approach bears certain resemblance to the notions of focus/adoption [5] and freeze/thaw [12] respectively in the literature. However, there is some fundamental difference lying in between. In Vault [5], the sharing (of an *adoptee*) enabled by an adoption is temporary (within the scope of the *adopter*). As a consequence, this design choice makes it difficult to support general sharing such as ML references, which can be arbitrarily aliased and passed around without any constraints. We can readily give some intuitive account of adoption/focus in our framework. Given two linear resources r_1 and r_2, adopting r_1 by r_2 corresponds to forming a combined resource $r_2' = r_1 \otimes r_2$ and focussing can be encoded as applying the prop $r_2' \rightarrow_i (r_1 \otimes (r_1 \rightarrow_l r_2'))$ to obtain/restore r_1 back and forth from/to r_2'. In L^3 [12], to prevent forming *re-thawing* a sharable object (similar to *double borrow* in our context), a notion named *thaw token* (which is linear) is adopted to keep track all the thawed objects. Although more flexible in theory, each thaw token must be threaded through every function that uses shared objects and the practicality of such an approach is yet to be shown.

In summary, we give a design in this paper to effectively promote the use of linear types in reasoning about resource usage in practical programming. We formalize this design in a type system where a modality is introduced to support safe resource sharing. In particular, we make use of a notion of *types with effects* [9] in overcoming the problem of double borrow. We also show some interesting and realistic programming examples including implementations of references and sharable arrays, which are all verified in ATS [20].

References

1. Asperti, A.: A logic for concurrency. Technical report, Dipartimento di Informatica, University of Pisa (1987)
2. Chen, C.: Type Inference in Applied Type System, 2005. PhD thesis, Boston University (September 2005)
3. Chen, C., Xi, H.: Combining Programming with Theorem Proving. In: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming, Tallinn, Estonia, September 2005, pp. 66–77 (2005)
4. Chirimar, J., Gunter, C.A., Riecke, G.: Reference Counting as a Computational Interpretation of Linear Logic. Journal of Functional Programming 6(2), 195–244 (1996)

5. Fahndrich, M., Deline, R.: Adoption and Focus: Practical Linear Types for Imperative Programming. In: Proceedings of the ACM Conference on Programming Language Design and Implementation, Berlin, June 2002, pp. 13–24 (2002)
6. Girard, J.-Y.: Linear logic. Theoretical Computer Science 50(1), 1–101 (1987)
7. Hofmann, M.: A type system for bounded space and functional in-place update. Nordic Journal of Computing 7(4), 258–289 (Winter 2000)
8. Igarashi, A., Kobayashi, N.: Garbage Collection Based on a Linear Type System. In: Preliminary Proceedings of the 3rd ACM SIGPLAN Workshop on Types in Compilation (TIC 2000) (September 2000)
9. Jouvelot, P., Gifford, D.K.: Algebraic reconstruction of types and effects. In: Proceedings of 18th ACM SIGPLAN Symposium on Principles of Programming Languages, January 1991, pp. 303–310 (1991)
10. Kobayashi, N.: Quasi-linear types. In: Proceedings of the 26th ACM Sigplan Symposium on Principles of Programming Languages (POPL 1999), San Antonio, Texas, USA, pp. 29–42 (1999)
11. Milner, R., Tofte, M., Harper, R.W., MacQueen, D.: The Definition of Standard ML (Revised). MIT Press, Cambridge (1997)
12. Morrisett, G., Ahmed, A., Fluet, M.: L^3: A Linear language with locations. In: Urzyczyn, P. (ed.) TLCA 2005. LNCS, vol. 3461, pp. 293–307. Springer, Heidelberg (2005)
13. Shi, R.: Types for Safe Resource Sharing in Sequential and Concurrent Programming. PhD thesis, Boston University (2007)
14. Shi, R., Xi, H.: A Linear Type System for Multicore Programming. In: Proceedings of Simposio Brasileiro de Linguagens de Programacao, Gramado, Brazil (August 2009)
15. Turner, D.N., Wadler, P.: Operational interpretations of linear logic. Theoretical Computer Science 227(1-2), 231–248 (1999)
16. Wadler, P.: Linear types can change the world. In: TC 2 Working Conference on Programming Concepts and Methods (Preprint), Sea of Galilee, pp. 546–566 (1990)
17. Walker, D., Watkins, K.: On Regions and Linear Types. In: Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP 2001), Florence, Italy, September 2001, pp. 181–192 (2001)
18. Xi, H.: Dependent Types in Practical Programming. PhD thesis, Carnegie Mellon University, pp. viii+189. (1998), `http://www.cs.cmu.edu/~hwxi/DML/thesis.ps`
19. Xi, H.: Applied Type System (extended abstract). In: Berardi, S., Coppo, M., Damiani, F. (eds.) TYPES 2003. LNCS, vol. 3085, pp. 394–408. Springer, Heidelberg (2004)
20. Xi, H.: Applied Type System (2005), `http://www.cs.bu.edu/hwxi/ATS`
21. Xi, H., Pfenning, F.: Dependent Types in Practical Programming. In: Proceedings of 26th ACM SIGPLAN Symposium on Principles of Programming Languages, San Antonio, Texas, January 1999, pp. 214–227. ACM Press, New York (1999)
22. Zhu, D., Xi, H.: Safe Programming with Pointers through Stateful Views. In: Hermenegildo, M.V., Cabeza, D. (eds.) PADL 2004. LNCS, vol. 3350, pp. 83–97. Springer, Heidelberg (2005)

Author Index