

George Eleftherakis
Mike Hinchey
Mike Holcombe (Eds.)

LNCS 7504

Software Engineering and Formal Methods

10th International Conference, SEFM 2012
Thessaloniki, Greece, October 2012
Proceedings



Springer

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Alfred Kobsa

University of California, Irvine, CA, USA

Friedemann Mattern

ETH Zurich, Switzerland

John C. Mitchell

Stanford University, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz

University of Bern, Switzerland

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

TU Dortmund University, Germany

Madhu Sudan

Microsoft Research, Cambridge, MA, USA

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max Planck Institute for Informatics, Saarbruecken, Germany

George Eleftherakis Mike Hinchey
Mike Holcombe (Eds.)

Software Engineering and Formal Methods

10th International Conference, SEFM 2012
Thessaloniki, Greece, October 1-5, 2012
Proceedings



Springer

Volume Editors

George Eleftherakis
The University of Sheffield Internal Faculty
CITY College, Computer Science Department
3, Leontos Sofou St.
546 26 Thessaloniki, Greece
E-mail: eleftherakis@city.academic.gr

Mike Hinchey
University of Limerick
Lero - the Irish Software Engineering Research Centre
T2-013
Limerick, Ireland
E-mail: mike.hinchey@lero.ie

Mike Holcombe
University of Sheffield
Department of Computer Science
Regent Court, 211 Portobello
Sheffield, S1 4DP, UK
E-mail: m.holcombe@dcs.shef.ac.uk

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-33825-0 e-ISBN 978-3-642-33826-7
DOI 10.1007/978-3-642-33826-7
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012947808

CR Subject Classification (1998): D.2.4, D.2, F.3, D.3, D.1.5, C.2, C.2.4

LNCS Sublibrary: SL 2 – Programming and Software Engineering

© Springer-Verlag Berlin Heidelberg 2012

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

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

Printed on acid-free paper

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

Preface

This volume contains the proceedings of the 10th International Conference on Software Engineering and Formal Methods (SEFM 2012) held October 1–5, 2012 in Thessaloniki, Greece. The conference was hosted by the University of Sheffield International Faculty, CITY College, and the South East European Research Centre (SEERC), under the auspices of the Macedonia Thrace Chapter of the Greek Computer Society (EPY).

The SEFM conference aspires to advance the state of the art in formal methods, to enhance their scalability and usability with regards to their application in the software industry, and to promote their integration with practical engineering methods. To this end, SEFM brings together practitioners and researchers from academia, industry, and government.

The Program Committee of SEFM 2012 received 98 full submissions from all over the world. Each paper was reviewed by at least three reviewers. The Program Committee selected 19 research papers, 2 tool papers, and 3 short papers for inclusion in this volume, based on review reports and discussions, together with 2 invited papers. The conference program included three keynote speakers: Cliff Jones (Newcastle University), Corrado Priami (University of Trento), and Wolfgang Reisig (University of Berlin).

The conference was preceded by a graduate school (September 24–28, 2012) and four satellite workshops (October 1–2, 2012) following the traditions of SEFM. The lectures were held by Antonio Cerone (UNU-IIST), Markus Roggenbach (Swansea University), Bernd-Holger Schlingloff (Humboldt University), Gerardo Schneider (University of Gothenburg), and Siraj Ahmed Shaikh (Coventry University). The workshops that joined the conference were the 3rd International Workshop on Formal Methods and Agile Methods (FM+AM 2012), the 6th International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2012), the 1st International Symposium on Innovation and Sustainability in Education (InSuEdu 2012), and the 1st International Symposium on Modelling and Knowledge Management for Sustainable Development (MoKMaSD 2012).

We would like to thank the invited speakers and all the authors for submitting their papers, as well as the Program Committee members and all the reviewers for their efforts in the selection process. We would also like to thank the school lecturers for accepting our invitation to present at the school. Finally, we are grateful to the members of the Steering Committee and the Organizing Committee, as well as everyone else whose efforts contributed to making the conference a success.

July 2012

George Eleftherakis
Mike Hinchey
Mike Holcombe

Program Committee

| | |
|------------------------|--|
| Bernhard K. Aichernig | TU Graz |
| Ade Azurat | Fasilkom UI |
| Luis Barbosa | Universidade do Minho |
| Thomas Anung Basuki | United Nations University |
| Fevzi Belli | University of Paderborn |
| Alexandre Bergel | University of Chile |
| Dimitar Birov | Sofia University |
| Jonathan P. Bowen | Museophile Limited |
| Christof J. Budnik | Siemens Corporate Research |
| Ana Cavalcanti | University of York |
| Antonio Cerone | United Nations University, UNU-IIST |
| Benoît Combemale | IRISA, Université de Rennes 1 |
| Anthony J. Cowling | University of Sheffield |
| Van Hung Dang | University of Engineering and Technology |
| Giannakopoulou Dimitra | NASA Ames |
| Dimitris Dranidis | University of Sheffield International Faculty, CITY College |
| George Eleftherakis | University of Sheffield International Faculty, CITY College |
| José Luiz Fiadeiro | University of Leicester |
| John Fitzgerald | Newcastle University |
| Martin Fränzle | Carl von Ossietzky Universität Oldenburg |
| Stefania Gnesi | ISTI-CNR |
| Klaus Havelund | Jet Propulsion Laboratory, California Institute of Technology |
| Rob Hierons | Brunel University |
| Mike Hinchey | Lero – The Irish Software Engineering Research Centre |
| Florentin Ipate | University of Pitesti |
| Jean-Marie Jacquet | University of Namur |
| Tomasz Janowski | UNU-IIST Center for Electronic Governance |
| Panagiotis Katsaros | Aristotle University of Thessaloniki |
| Petros Kefalas | South-East European Research Centre (SEERC) |
| Joseph Kiniry | IT University of Copenhagen |
| Martin Leucker | University of Lübeck |
| Peter Lindsay | The University of Queensland |
| Zhiming Liu | United Nations University, Macau SAR, China |
| Antónia Lopes | University of Lisbon |
| Tiziana Margaria | University Potsdam |
| Mercedes Merayo | Universidad Complutense de Madrid |
| Stephan Merz | INRIA Lorraine |
| Marius Minea | Polytechnic University of Timisoara |
| Mizuhito Ogawa | Japan Advanced Institute of Science and Technology |

| | |
|-------------------|---|
| Olaf Owe | University of Oslo |
| Gordon Pace | University of Malta |
| Anna Philippou | University of Cyprus |
| Ernesto Pimentel | University of Malaga |
| Sanjiva Prasad | Indian Insitute of Technology Delhi |
| Anders Ravn | Aalborg University |
| Wolfgang Reisig | Humboldt University of Berlin |
| Leila Ribeiro | Universidade Federal do Rio Grande do Sul |
| Bernhard Rumpe | RWTH Aachen University |
| Augusto Sampaio | Federal University of Pernambuco |
| Ina Schaefer | Technical University Braunschweig |
| Gerardo Schneider | Chalmers, University of Gothenburg |
| Joseph Sifakis | VERIMAG |
| Massimo Tivoli | University of L'Aquila |
| Viktor Vafeiadis | MPI-SWS |
| Husnu Yenigun | Sabanci University |

Additional Reviewers

| | |
|----------------------|-----------------------|
| Barais, Olivier | Hüttel, Hans |
| Bordihn, Henning | Iliasov, Alexei |
| Bryans, Jeremy W. | Iyoda, Juliano |
| Butterfield, Andrew | Jeannet, Bertrand |
| Böde, Eckard | Jobstl, Elisabeth |
| Bøgholm, Thomas | Kapitsaki, Georgia |
| Cadavid, Juan | Kemper, Stephanie |
| Chen, Zhenbang | Kim, Soon-Kyeong |
| Colombo, Christian | Kromodimoeljo, Sentot |
| Cornélio, Márcio | Lochau, Malte |
| Cubo, Javier | Lorber, Florian |
| Dan, Li | Magazinius, Ana |
| Decker, Normann | Mazzanti, Franco |
| Ellen, Christian | Mazzara, Manuel |
| Faber, Johannes | Mehta, Farhad |
| Florian, Mihai | Meredith, Patrick |
| Francalanza, Adrian | Micallef, Mark |
| Garrido, Daniel | Minamide, Yasuhiko |
| Gerwinn, Sebastian | Müller, Richard |
| Gierds, Christian | Müllner, Nils |
| Gonnord, Laure | Nakajima, Shin |
| Gutiérrez, Francisco | Nunes, Isabel |
| Günther, Henning | Nuñez, Manuel |
| Hansen, Rene Rydhof | Ogata, Kazuhiro |
| Havsá, Irene | Okikka, Joseph |
| Holzmann, Gerard | Payne, Richard |

| | |
|--------------------------|------------------|
| Pohlmann, Christian | Stenzel, Kurt |
| Prisacariu, Cristian | Stolz, Volker |
| Rouquette, Nicolas | Sürmeli, Jan |
| Sannier, Nicolas | Terauchi, Tachio |
| Santos, André L. | Thoma, Daniel |
| Schoeberl, Martin | Thüm, Thomas |
| Schönfelder, Rene | Tiran, Stefan |
| Seki, Hiroyuki | Turker, Uraz |
| Skou, Arne | Valencia, Frank |
| Spagnolo, Giorgio Oronzo | Wen, Lian |
| Spoto, Fausto | Winter, Kirsten |
| Stefanescu, Alin | Yuen, Shoji |

Sponsors

- EPY (Greek Computer Society, Macedonia-Thrace Chapter, Greece)
- CITY College (International Faculty of the University of Sheffield, Greece)
- SEERC (South-East European Research Center, Greece)

Table of Contents

Keynote Talks

| | |
|---|----|
| Abstraction as a Unifying Link for Formal Approaches to Concurrency | 1 |
| <i>Cliff B. Jones</i> | |
| A Rule-Based and Imperative Language for Biochemical Modeling and Simulation | 16 |
| <i>Durica Nikolić, Corrado Priami, and Roberto Zunino</i> | |

Regular Papers

| | |
|--|-----|
| Sound Control-Flow Graph Extraction for Java Programs with Exceptions | 33 |
| <i>Afshin Amighi, Pedro de C. Gomes, Dilian Gurov, and Marieke Huisman</i> | |
| Checking Sanity of Software Requirements | 48 |
| <i>Jiří Barnat, Petr Bauch, and Luboš Brim</i> | |
| TVAL+ : TVLA and Value Analyses Together | 63 |
| <i>Pietro Ferrara, Raphael Fuchs, and Uri Juhasz</i> | |
| A Systematic Approach to Atomicity Decomposition in Event-B | 78 |
| <i>Asieh Salehi Fathabadi, Michael Butler, and Abdolbaghi Rezazadeh</i> | |
| Compositional Reasoning about Shared Futures | 94 |
| <i>Crystal Chang Din, Johan Dovland, and Olaf Owe</i> | |
| Verification of Aspectual Composition in Feature-Modeling | 109 |
| <i>Qinglei Zhang, Ridha Khedri, and Jason Jaskolka</i> | |
| A Denotational Model for Instantaneous Signal Calculus | 126 |
| <i>Yongxin Zhao, Longfei Zhu, Huibiao Zhu, and Jifeng He</i> | |
| A Timed Mobility Semantics Based on Rewriting Strategies | 141 |
| <i>Gabriel Ciobanu, Maciej Koutny, and Jason Steggle</i> | |
| Towards a Formal Component Model for the Cloud | 156 |
| <i>Roberto Di Cosmo, Stefano Zacchiroli, and Gianluigi Zavattaro</i> | |
| The Rely/Guarantee Approach to Verifying Concurrent BPEL Programs | 172 |
| <i>Huibiao Zhu, Qiwen Xu, Chris Ma, Shengchao Qin, and Zongyan Qiu</i> | |

| | |
|---|-----|
| Completing the Automated Verification of a Small Hypervisor – Assembler Code Verification | 188 |
| <i>Wolfgang Paul, Sabine Schmaltz, and Andrey Shadrin</i> | |
| A Configuration Approach for IMA Systems | 203 |
| <i>Visar Januzaj, Stefan Kugele, Florian Biechele, and Ralf Mauersberger</i> | |
| polyLARVA: Runtime Verification with Configurable Resource-Aware Monitoring Boundaries | 218 |
| <i>Christian Colombo, Adrian Francalanza, Ruth Mizzi, and Gordon J. Pace</i> | |
| Frama-C: A Software Analysis Perspective | 233 |
| <i>Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski</i> | |
| An Optimization Approach for Effective Formalized fUML Model Checking | 248 |
| <i>Islam Abdelhalim, Steve Schneider, and Helen Treharne</i> | |
| Efficient Probabilistic Abstraction for SysML Activity Diagrams | 263 |
| <i>Samir Ouchani, Otmane Ait Mohamed, and Mourad Debbabi</i> | |
| ML Dependency Analysis for Assessors | 278 |
| <i>Philippe Ayrault, Vincent Benayoun, Catherine Dubois, and François Pessaux</i> | |
| An Improved Test Generation Approach from Extended Finite State Machines Using Genetic Algorithms | 293 |
| <i>Raluca Lefticaru and Florentin Ipate</i> | |
| Securely Accessing Shared Resources with Concurrent Constraint Programming | 308 |
| <i>Stefano Bistarelli and Francesco Santini</i> | |

Short Papers

| | |
|--|-----|
| A Practical Approach for Closed Systems Formal Verification Using Event-B | 323 |
| <i>Brett Bicknell, Jose Reis, Michael Butler, John Colley, and Colin Snook</i> | |
| Extensible Specifications for Automatic Re-use of Specifications and Proofs | 333 |
| <i>Daniel Matichuk and Toby Murray</i> | |
| Implementing Tactics of Refinement in CRefine | 342 |
| <i>Madiel Conserva Filho and Marcel Vinicius Medeiros Oliveira</i> | |

Tool Papers

| | |
|---|-----|
| JSXM: A Tool for Automated Test Generation | 352 |
| <i>Dimitris Dranidis, Konstantinos Bratanis, and Florentin Ipat</i> | |
| A Low-Overhead, Value-Tracking Approach to Information Flow Security | 367 |
| <i>Kostyantyn Vorobyov, Padmanabhan Krishnan, and Phil Stocks</i> | |
| Author Index | 383 |

Abstraction as a Unifying Link for Formal Approaches to Concurrency

Cliff B. Jones

School of Computing Science, Newcastle University, NE1 7RU, UK
cliff.jones@ncl.ac.uk

Abstract. Abstraction is a crucial tool in specifying and justifying developments of systems. This observation is recognised in many different methods for developing sequential software; it also applies to some approaches to the formal development of concurrent systems although there its use is perhaps less uniform. The rely/guarantee approach to formal design has, for example, been shown to be capable of recording the design of complex concurrent software in a “top down” stepwise process that proceeds from abstract specification to code. In contrast, separation logics were—at least initially—motivated by reasoning about details of extant code. Such approaches can be thought of as “bottom up”. The same “top down/bottom up” distinction can be applied to “atomicity refinement” and “linearisability”. Some useful mixes of these approaches already exist and they are neither to be viewed as competitive approaches nor are they irrevocably confined by the broad categorisation. This paper reports on recent developments and presents the case for how careful use of abstractions can make it easier to marry the respective advantages of different approaches to reasoning about concurrency.

1 Introduction

There is much research activity around formal support for concurrency. The reasons for this ought be clear. For non-critical applications, good (semi-formal) engineering methods are sometimes adequate for sequential programs. Such methods borrow much from past formal research and, even here, organisations such as Praxis report that adding formal methods to the development process can bring about a return on investment because of the tighter control and reduction in the late discovery of errors that are expensive to fix because they result from decisions made much earlier in the design process.

Once one moves to the design of concurrent systems, the enormous increase in the number of execution paths brought about by thread interaction makes it effectively impossible to have any confidence in correctness without some form of formal proof.

One might ask why designers should be so rash as to venture into such dangerous territory. Unfortunately, there is no choice — the pressures to face concurrency become ever greater. First, the (economic) limits for the extrapolation of “Moore’s law” mean that hardware performance can only be increased by moving from “multi-core” to “many-core” hardware (i.e. numbers of threads likely to measured in hundreds). Secondly, embedded systems often run in parallel with physical phenomena that are varying continuously; control software linked to the physical world by sensors and actuators

cannot ignore these state changes. Thirdly, a class of application has to be implemented by physically distributed sets of processors.

The combination of a realisation that concurrency cannot be avoided with the acknowledgement that its mastery requires formal tools has generated many research strands. Notable activity in the areas of rely/guarantee thinking, separation logic, atomicity refinement and linearisability is addressed in the body of this paper (citations to relevant papers are given below). To apply some of these research ideas to the paper itself, the attempt here is to look for constructive interaction between several threads of research. In particular, this paper looks to tease out the key *concepts* from the various methods and indicate a path to one or more methods that achieve real synergy from what are currently rather distinct approaches. This is a much deeper exercise than just seeking combinations of notations.

A key distinction between top-down and bottom-up approaches is used below. Any such dichotomy must be viewed with care and there is certainly no intention to make a judgement that one approach is “better” than the other. If the task is to improve the quality of millions of lines of legacy code, there is little choice but to use bottom-up methods. On the other hand, faced with the challenge of developing and documenting a large system from scratch, it would be unwise not to record each stage of design “from the top” (i.e. the specification).¹ One particular form of top-down formal development that makes solid engineering sense is known as “posit and prove”. The idea is that each step of design starts by recording an engineering intuition that might be a decomposition of a problem into sub-tasks or the choice of a data representation for something that was previously an abstraction that achieved brevity. In suitable formal methods, such a posited step gives rise to “proof obligations” whose discharge justifies the correctness of the step. It is well understood that redundancy is essential for dependability and this posit and prove approach provides constructive redundancy. As discussed below, there is a technical requirement of “compositionality” for such methods. Although relatively easy to achieve for sequential systems, compositionality is far more elusive in the world of concurrent systems.

It is however important to remember that no judgement is being made here about the relative merits of top-down and bottom-up approaches. There is indeed evidence that, in (complex) bottom-up analysis, it is necessary to recreate abstractions that are hidden in the code [9,39]. It is hoped, and anticipated, that any new methods devised from –for example– the combination of concepts from separation logic and rely/guarantee thinking will provide benefit to both top-down and bottom-up approaches.

1.1 Rely/Guarantee Thinking

Specifications of sequential programs are normally given as pre and post conditions.² Floyd showed [16] how predicate calculus assertions could be added to a flowchart of

¹ Of course, there exists the issue of how to obtain the starting specification. This is not the subject of the current paper but some contribution to a resolution of this issue is made in [36]. Interestingly, this joint work with Ian Hayes and Michael Jackson uses rely/guarantee ideas.

² This paper does not waste space making the case for formality in system design but it is worth remembering that formal concepts (e.g. data type invariants) are useful even when used in specifications and developments that are not completely formal.

a program to present a proof that the program satisfied a specification; Hoare made the essential step [23] to give an inference system for asserted texts. Few programs can tolerate completely arbitrary starting states and it is important to note that pre conditions effectively grant a developer assumptions about the starting states in which the created software will be deployed; in contrast, post conditions are requirements on the running code.

The essence of concurrency is *interference*. In shared variable concurrency, such interference manifests itself by one thread having to tolerate changes being made to its state by other processes.³ No useful program can achieve a sensible outcome in the presence of completely arbitrary interference. This is recognised in the rely/guarantee approach [28,29,30] by recording the acceptable interference as a relation—known as a rely condition—over pairs of states. (The use of relations fits with the fact that VDM [27] employs relational post conditions.) Like pre conditions, rely conditions can be seen as permission for the developer to make assumptions about contexts in which the final code will be deployed. The commitment as to what interference a running component will impose on its neighbours is recorded in a guarantee condition (again a relation over states).

Not surprisingly, the proof obligations required in rely/guarantee reasoning are more complex than for sequential programs. There is also scope for more variability and the proof obligations concerned with introducing concurrency differ over various publications. One form (geared to decomposition — see Sect. 2.1) is:

$$\begin{array}{c}
 \{P, R_l\} s_l \{G_l, Q_l\} \\
 \{P, R_r\} s_r \{G_r, Q_r\} \\
 R \vee G_r \Rightarrow R_l \\
 R \vee G_l \Rightarrow R_r \\
 G_l \vee G_r \Rightarrow G \\
 \hline
 \boxed{\text{Par-I}} \frac{\overline{P} \wedge Q_l \wedge Q_r \wedge (R \vee G_l \vee G_r)^* \Rightarrow Q}{\{P, R\} s_l \parallel s_r \{G, Q\}}
 \end{array}$$

What is crucial is that rely/guarantee systems of rules can be made “compositional” in the same sense that Hoare-like methods for sequential programs enjoy this essential property: a specification with rely/guarantee conditions records all that a developer need know to create acceptable code. De Roeper’s exhaustive survey book [10] distinguishes between compositional and non-compositional development methods pointing out that the *post-facto* “Einmischungsfrei” proof obligation in the Owicki/Gries method makes it non-compositional.

Examples of rely/guarantee development are postponed to Section 2 where future issues are explored. The most approachable text on past research in this area might be [31]; a proof of the soundness of rely/guarantee methods is given in [7].

Compositionality is key to methods that are to be used in a *top-down* style where a development is started from an overall specification and decomposed step by step until the finest sub-components have been developed into code. Data abstraction is key

³ Process algebras might appear to finesse the whole issue of “states” but processes can be constructed that effectively store values that can be changed and read by interaction; the issue of interference reappears as reasoning about the traces of such interaction.

to achieving brief and understandable specifications; the corresponding development method of *data reification* is also compositional. Interaction between data reification and rely/guarantee thinking is explored in Section 3.1 below.

1.2 Separation Logic(s)

Just as in the preceding section, it is neither the aim to offer a complete description nor to present a full history of research on separation logics⁴ and the extension to concurrent separation logic here. Some important milestones include [6,46,44,51,26,52,45,4,47]. In fact, Peter O’Hearn pointed out at the Cambridge meeting to mark Tony Hoare’s 75th birthday that the fundamental idea of disjoint parallelism dates back to [25]. Of interest for the current paper is the notion of “separating conjunction” and the emphasis on reasoning about heap variables.⁵

To show that execution threads do not “race” on access to a particular variable, it is enough to establish that there is mutual exclusion between any reference from those threads to the relevant variable. With standard (“stack”) variables, this can be achieved by ensuring that each variable is visible to at most one thread. Separation logic is, however, more often applied to programs using dynamic (“heap”) variables and it offers ways of reasoning about the dynamic “ownership” of their addresses. The principal tool is the “separating conjunction”: if two conjuncts have disjoint frames, they can be associated with different threads of execution. Thus, two parallel processes can achieve a post condition written as a separating conjunction by each achieving one of the conjuncts. There is no interference and no “race” on addresses.

One important aspect of separation logic is that ownership can change dynamically; this does however appear to be the reason that the “frame” of an operation is determined by the alphabet of its assertions. Perhaps more significant for the objectives of the current paper is that there are—in addition to separating conjunction—a number of other operators in separation logic and that all of the operators are linked by useful algebraic laws. A practical point is that the researchers involved with separation logic have put considerable effort into providing tool support for their ideas.

The majority of papers on separation logic focus on “heap” variables⁶ that can be dynamically allocated and freed. The examples chosen are typically of low-level (operating system like) code performing tasks like maintaining concurrent queues and delicate manipulations of tree representations. This creates the impression that separation logic is aimed at “bottom up” analysis of extant code. Furthermore, much of the commendable effort on tool support is aimed at establishing freedom from stated faults such as race conditions in extant code.

As an example (that is useful in Section 2) Reynolds considers a sequential in place list reversal in [52]; the introduction of the problem is:

⁴ The justification for using the plural of “logic” is [49].

⁵ In [50] a move to high level programming constructs is made — that paper does not, however, link the constructs to concurrency.

⁶ In [48], the ideas of separation logic are applied to “stack” variables but the resulting system appears less pleasing than that for heap variables.

The following program performs an in-place reversal of a list:

$$j := \mathbf{nil}; \mathbf{while} \ i \neq \mathbf{nil} \ \mathbf{do} \\ (k := [i + 1]; [i + 1] := j; j := i; i := k).$$

(Here the notation $[e]$ denotes the contents of the storage at address e .)

The reasoning then employs “separating conjunction” ($*$) as in

$$\exists \alpha, \beta \cdot \mathit{list}(\alpha, i) * \mathit{list}(\beta, j)$$

to specify that the lists starting respectively at addresses i and j encompass separate sets of addresses. The extremely succinct separation logic rule for a parallel construct is

$$\boxed{\text{SL}} \frac{\frac{\{P_l\} \ s_l \ \{Q_l\}}{\{P_r\} \ s_r \ \{Q_r\}}}{\{P_l * P_r\} \ s_l \ || \ s_r \ \{Q_l * Q_r\}}$$

There is a lot going on in this compact rule. The separating conjunction (written as an infix “*” operator) is only valid if the frames of the two disjuncts are disjoint. Moreover, since there is no explicit declaration of the read/write frames of either operand, these are determined by the alphabets of the expressions.⁷ It is also the norm that the *SL* rule is used on heap variables (i.e. machine addresses that are allocated at run time rather than names of variables that are translated to machine addresses by a compiler).

Before considering the potential for using the core ideas of such ownership logics early in the design process, the next section reviews what has already been done to obtain complementary benefits from the two approaches outlined.

1.3 Existing Complementarity

The research around rely/guarantee thinking and separation logics is extremely active. Both [57] and Viktor Vafeiadis’ Cambridge thesis [56] propose a combination of rely/guarantee and separation thinking: the “RGSep” rules neatly specialise to either of the original sets of rules.⁸

$$\boxed{\text{RGSep}} \frac{\frac{\{P_l, R \cup G_r\} \ s_l \ \{G_l, Q_l\}}{\{P_r, R \cup G_l\} \ s_r \ \{G_r, Q_r\}}}{\{P_l * P_r, R\} \ s_l \ || \ s_r \ \{G_l \cup G_r, Q_l * Q_r\}}$$

(The brevity of this rule is slightly artificial: see discussion in Sect. 2.1.)

Matt Parkinson’s “Deny/Guarantee” system [14] extends rely/guarantee ideas to cope with the dynamic forking of threads. Although it belongs to a later discussion, it is also worth mentioning here the “RGSim” approach [40].

⁷ This has the surprising consequence that some expression E and $E \wedge x = x$ do not have an equivalent effect.

⁸ Xinyu Feng’s research on SAGL [15] goes in a similar direction to RGSep — for reasons of space, discussion here is confined to the latter.

An interesting trace of interaction between the two main approaches discussed so far can be seen in a series of papers that all address “Asynchronous Communication Methods” in general — and more specifically Hugo Simpson’s “4-slot” algorithm [54]. Richard Bornat is a key figure in Separation Logic research but both of his papers [2,3] make use of rely/guarantee descriptions — a fact that is explicit even in the title of the earlier contribution. The current author’s contributions [37,38] interleave in time with Bornat’s and throw an interesting light on a distinction that has been drawn between methods. The talk by Peter O’Hearn at the 2005 MFPS (published as [45]) suggested that the natural tool for proving the absence of data races is separation logic — in contrast, rely/guarantee reasoning is appropriate for “racy” programs. One key feature of Simpson’s 4-slot algorithm is the avoidance of clashes (or data races) on any of the four slots used as an interface between the entirely asynchronous read/write processes. However, [38] uses rely/guarantee conditions to describe the non-interference at an abstract level — in fact, this is done before the number of slots has been determined.

The MFPS categorisation *can be* useful but it is important to remember that any such split must be used judiciously. (This warning applies, of course, also to the use of “top down” versus “bottom up” in the current paper.)

2 Seeking Further Synergy

It is clear that neither rely/guarantee nor separation logic alone can cope with all forms of concurrency reasoning. This is precisely the reason that looking for synergy is worthwhile. Lacunae on the rely/guarantee side certainly include the ability to reason about (dynamic) ownership and discourses about heap variables. (The extent to which the ideas of Sect. 2.4 below can finesse these gaps is yet to be determined.)

From the other side, there are concepts with which separation logic appears to struggle because interference can be problematic without races. Consider the innocent looking thread, say α :

$$x \leftarrow 1; y \leftarrow x$$

Suppose that some thread control variables are added so that parallel processes do not interfere during the execution of either assignment in thread α . The classical Hoare rule would carry the information that x has the value 1 to the precondition of the second assignment. If there is a concurrent process that increases the value of x by an arbitrary amount (e.g. a loop that continues to execute $x \leftarrow x + 1$), then this transfer of “knowledge” is certainly invalid. There are no data races here but, if one is to conclude $y \geq 1$, there have to be assumptions (rely conditions) about the context in which thread α will run.

This section reports on some on-going research and speculates about some further directions where abstraction might make it possible to get underneath the syntactic details of both rely/guarantee and separation logic; the hope is that, by really understanding the conceptual contribution, one or more methods will evolve that are intuitive to the intended users.

2.1 Algebraic Laws about Rely/Guarantee

It has never been the intention to fix on one set of proof rules for rely/guarantee reasoning. One cause of differences between rules is the distinction between rules that are convenient for composition versus those best suited to decomposition. This distinction can be illustrated even on the standard Hoare-rule for **while** constructs: the most common form of the rule gives the post-condition of a **while** as the conjunction of the loop invariant (say P) with the negation of the test condition written in the loop construct — thus $P \wedge \neg b$. This is a useful composition rule but it is unlikely that, when faced with a design step, the post condition will fall neatly into such a conjunction. An equivalent Hoare rule with an arbitrary post condition of Q to be achieved after the loop requires an additional hypothesis that $P \wedge \neg b \Rightarrow Q$.⁹ For pre/post conditions, this distinction is small and often glossed over in texts but for more complex rely/guarantee specifications, the difference in presentation between composition and decomposition rules is greater — as can be seen by contrasting the rules in Sects. 1.1/1.3. Of course, the rules are related by suitable “weakening rules” but the choice of an appropriate form of the rule does matter when providing tool support for proof obligation generation.

There are also more interesting differences between versions of rules that generate proof obligations for rely/guarantee reasoning. In [8], for example, an “evolution invariant” is used that can be thought of as relating the initial state to any state that can arise. Just as (standard, single state) data type invariants have proved useful intuitive aids in developments that are not necessarily formal, the idea of evolution invariants has sparked interest in a number of areas.

These points prompted a desire to find something more basic that could be used to reason about interference in a rely/guarantee style. In [18], inspiration is drawn from the refinement calculus as presented by Carroll Morgan [41] (and, in particular the “invariant command” of [42]) to employ **guar** and **rely** constructs written as commands. The move away from fixed format presentation of rely/guarantee as in

$$\{P, R\} s \{G, Q\}$$

brings advantages similar to those of the refinement calculus over Hoare-triples.¹⁰ In addition, it makes clear that there is an algebra of the clauses. For example, the trading of clauses between guarantee and post conditions that appears almost as black magic in earlier papers becomes a law

$$(\mathbf{guar} G \bullet [Q \wedge G^*]) = (\mathbf{guar} G \bullet [Q])$$

Furthermore, the collection of laws in [18] fits with a pleasing refinement calculus top-down development style (the reader is referred to that paper for the full set of rules and a worked example — the style of formal semantics used there is that of [21]).

⁹ VDM uses relational post conditions that make it possible to express termination (sometimes referred to as “total correctness”) via well-founded relations — this feels more natural than the “variant function” of [11]. Be that as it may, the same distinction between composition and decomposition presentations remains.

¹⁰ Jürgen Dingel has also looked at presenting rely/guarantee ideas in a form of refinement calculus setting — his objective in [13,12] was not however to separate the commands in the way done in [18].

2.2 Framing

The early papers on rely/guarantee reasoning used VDM’s keyword style to define the **rd/wr** frames. The move to a refinement calculus presentation not only gives a more linear notation, it also prompts the use of a compact notation to specify the write frame of a command. Thus:

$$x: [Q]$$

requires that the relational post condition Q is achieved with changes only being made to the variable x . This makes a small step towards the compact notation of separation logic. Rather than go to the complete determination of frames from the alphabets of assertions used there, a sensible intermediate step might be to write pre and post conditions as predicates with explicit parameter lists and have the arguments of the former determine the read frame and the extra parameters of the latter determine the write frame. The indirection of having named predicates would pose little overhead in large applications because it is rarely practical to write specifications in a single line.

2.3 Possible Values

Another interesting development in [38] is the usefulness of being able, in assertions, to discuss the “possible values” that a variable can take. This idea actually arose from a flaw in an earlier version of our development of Simpson’s four-slot implementation of *Asynchronous Communication Mechanisms*: at some point there was a need to record in a post condition for a *Read* (sub-)operation that the variable $hold-r$ acquired the value from a variable $fresh-w$ that could be set by a *Write* process. This was written in the earlier, flawed, version of the development [37] as $hold-r = \overline{fresh-w} \vee hold-r = fresh-w$. But allowing that $hold-r$ acquired the initial or final values of $fresh-w$ is not enough because the sibling (*Write*) process could execute many assignments to $fresh-w$ whilst the *Read* process was executing. This prompted a special notation for the set of values that can arise and the post condition of the *Read* process can be correctly recorded as $hold-r \in \widehat{fresh-w}$. The possible values notation is equally useful in, say, guarantee conditions and the full payoff comes in proofs.

An encouraging sign for the utility of the possible values notation (\widehat{x}) is that many other uses have been found for the same concept. Furthermore, a pleasing link with Ian Hayes’ on-going research on non-deterministic expression evaluation is formalised in [20].

2.4 Separation as an Abstraction

Thus far, several ways in which abstraction can facilitate both cleaner developments and, more generally, useful concepts for developing programs have been shown. In this more speculative section, a way of viewing the core concept of separation *as an abstraction* is explored.

Returning to Reynolds’ example of reversing a sequence, a top-down development might start with a post-condition built around the function $rev: X^* \rightarrow X^*$

$$rev(s) \triangleq \mathbf{if} \ s = [] \ \mathbf{then} \ s \ \mathbf{else} \ rev(\mathbf{tl} \ s) \overset{\curvearrowright}{\sim} [\mathbf{hd} \ s] \ \mathbf{fi}$$

The post condition itself only has to require that some variable, say s , is changed so that

$$r, s: [r = rev(\overleftarrow{s})]$$

(Notice that this specification gives the designer the permission to overwrite the variable s .)

This can be achieved by the following abstract program:

```

r ← [];
while s ≠ [] do
  r, s: [r = [hd  $\overleftarrow{s}$ ]  $\overset{\curvearrowright}{\sim}$   $\overleftarrow{r}$  ∧ s = tl  $\overleftarrow{s}$ ]
  {r  $\overset{\curvearrowright}{\sim}$  rev(s) = rev( $\overleftarrow{s}$ )}
od

```

Thus far, s and r are assumed to be distinct variables. That they are separate is a useful and natural abstraction. A design *decision* to choose a representation in which both variables are stored in the same vector must maintain the essential points of that abstraction! The requirement to maintain the abstraction of separation thus moves to a data reification step. It is yet to be worked out what form of separation logic best suits this view but it is hoped that it will again be a step towards combining the advantages of separation logic thinking with ideas from rely/guarantee and data reification. (Sect. 3.1 describes existing links between rely/guarantee reasoning and data reification.)

2.5 Fiction of Atomicity as an Abstraction for Linearisability

There is insufficient space here to go into a complete exploration of a further pair of approaches but it is worth mentioning that there are other issues that fit the analysis of two ideas that have evolved from top-down and bottom-up views and look ripe for reconsideration.

Research on linearisability was put on a firm foundation by [22]; recent interesting papers include [17,5]. The basic idea is to look at detailed sub-steps and to find a larger atomic operation that would have the same effect.

The idea that it is possible, in a top-down design process, to use a “fiction of atomicity” is discussed in [32,33] (for the origins of the ideas see references in these papers). The development process that links the abstraction to its realisation is known as “atomicity refinement” (or “splitting (software) atoms safely”). In one particular version of this process, equivalences were found that justified enhanced concurrency. What was crucial to the justification of these equivalences (see, for example, [53]) was a careful analysis of the language in which observations can be made. (To make the point most simply, if the observation language can observe timings, parallel processes are likely to be seen as running faster; but there are much more subtle dependencies to be taken into account as well.)

It must again be worthwhile to look at how these top-down and bottom-up views of varying the level of atomicity of processes can benefit from each other. Furthermore,

both the basic idea of separate sets of addresses and of rely/guarantee-like assumptions about the effect of the processes look likely to be important when reasoning about the different granularities.

3 Further Observations on Abstraction

Much is being made about the virtues of “abstraction” in this paper. It is useful to look both at some past successes and issues around the use of this panacea in software design. Section 2 starts out with a confession that the ideas in that section are speculative; it is likely that “issues” will arise in their development and that past experience might be useful in their resolution.

3.1 Rely/Guarantee Thinking and Data Reification

It is difficult to exaggerate the importance of data abstraction in specifying computer systems. Whilst it is true that there are cases like sorting where a post condition is much easier to write than an algorithm, most complex computing tasks can only be described in a brief and understandable way if their description is couched in terms of abstract objects that match the problem in hand. Of course, mathematical abstractions¹¹ are not necessarily available in programming languages. The top-down, stepwise, process of “reifying” abstractions to data types that are available in implementation languages is a key tool of formal methods (one of the first books to emphasise this is [27] and it has been given its due prominence in VDM ever since).

Because the current author had seen this importance, it was completely natural that—even in the earliest rely/guarantee developments—data abstraction and reification were deployed. What was less apparent was the strength of the link between the ideas. In fact, it was not until [33] that full recognition was given to the extent to which the ability to find a representation affected whether or not granularity assumptions could be met without locking. Having noticed this—on a range of examples—it can now be used to help guide the choice of rely and guarantee predicates.

3.2 What Happens When Abstraction Fails?

It is illuminating to sketch the history of data abstraction/reification in VDM.¹² Peter Lucas’ first proof of the equivalence of two distinct (VDL) operational semantic formulations in fact used a “twin machine” idea that amounted to a relation between the two models. It was only later that the research on VDM focused on the use of a “retrieve” function that was in effect a homomorphism from the representation back to the abstraction. This idea became the standard in VDM (e.g. [27]) and there was even a test devised for “implementation bias”. Everything was rosy in the abstraction garden until a few contrary examples appeared where, in each, it was impossible to find an unbiased

¹¹ It is interesting to remember that all of the formal specification notations VDM [27], Z [19] and B [1] employ the same collection of abstract objects: sets, sequences, maps and some form of record.

¹² This is done more fully, and in the context of other research, in [35].

state that covered all allowable implementations. The essence of the problem was that, for these rare applications, the abstract state had to contain information that allowed non-determinacy; but once design decisions removed the non-determinacy, the states could be simplified in a way that meant they had *less* information than the abstraction; this meant that no homomorphism could be found.

The problem of justifying such design decisions was overcome by adding a new data reification rule to VDM that derives from the research of Tobias Nipkow [43] (a parallel development in Oxford led to [24]). The essential point here is that one should strive for “bias freedom” but that it might not always be attainable. If this is genuinely the case, there may be a need to devise new proof methods.

One particular “trick” that is often used in reasoning about concurrent programs is to add “auxiliary” (or “ghost”) variables that record information that is not (readily) available in the actual variables of a program. The temptation to do this is often strong but this author has doubts about the wisdom of giving in to it. The danger is that it is difficult to put precise limits on what it is legitimate to record in ghost variables. Compositionality can be completely destroyed by recording information about a thread that one might wish to revise without changing the design of any concurrent threads. More is said on this topic in [34].

4 Conclusions and Next Steps

Clearly, much remains to be done to bring about intellectual and software tools that will contribute to the work of software design for concurrent systems.

In some senses, what marks out a useful formal method is not its ability to express *anything* but rather its expressive weakness! One seeks a notation that can cover a useful class of applications but be weak enough to be tractable. For some applications rely/guarantee conditions –coupled with liberal amounts of abstraction– fit this pattern. The basic rely/guarantee relations make it possible to go through a top-down development of non-trivial algorithms that allow (at least abstract) races on variable access. Ketil Stølen showed in [55] that the same basic framework can be extended to handle progress arguments. Similarly, separation logic approaches employ a notation for which useful tool support has been developed. What one has to seek is a sweet spot where much can be handled with a (close to) minimum of formal overhead.

Although the current author finds compact notations attractive (and remember the point made by Christopher Strachey that it is far easier to manipulate a string of symbols that fit on a line than multi-line texts), it is unavoidable that specifications of large systems get recorded using long formulae. This author has written many papers extolling the advantages of abstraction but has seen enough formal specifications of systems such as “cruise control” to avoid setting “single line specifications” as an objective. What is important is to have notations whose operators are linked with useful algebraic properties: separation logic clearly achieves this and, for rely/guarantee, [18] makes a first step in this direction but the objective must be kept in mind.

Another area where separation logic researchers have been wise is in their emphasis on tool support for their ideas. This must –and will– be an objective of our research to bring together different approaches to concurrency reasoning.

It is sadly the case that most currently available programming languages are poor vehicles for expressing (safe) concurrency. There is, therefore, a temptation to plan to embody the ideas from the research adumbrated in the body of this paper into yet-another programming language. Such is not the immediate objective of the current author who has seen too many languages that offer at most one new idea but implement many other concepts less well than existing languages. The first step is tractable design concepts (that might be used to develop programs into patterns in existing languages); it would be pleasing if these patterns were adopted by some careful language designer(s).

Acknowledgements. It is a pleasure to thank many research collaborators. The most relevant and active contacts on the research reported here are Ian Hayes and Rob Colvin (particularly Sect. 2.1); Matt Parkinson, Viktor Vafeiadis and Richard Bornat (particularly Sect. 1.3); Hongseok Yang and Alexey Gotsman (particularly Sect. 2.5); and all attendees at the productive series of concurrency meetings held in London, Cambridge, Newcastle and Dublin (and Oxford in July is eagerly anticipated).

The author of this paper gratefully acknowledges the funding for his research from the EPSRC Platform Grant TrAmS-2.

References

1. Abrial, J.-R.: *The B-Book: Assigning programs to meanings*. Cambridge University Press (1996)
2. Bornat, R., Amjad, H.: Inter-process buffers in separation logic with rely-guarantee (2010)
3. Bornat, R., Amjad, H.: Explanation of two non-blocking shared-variable communication algorithms. *Formal Aspects of Computing*, 1–39 (2011)
4. Brookes, S.D.: A semantics of concurrent separation logic. *Theoretical Computer Science* 375(1-3), 227–270 (2007)
5. Burckhardt, S., Gotsman, A., Musuvathi, M., Yang, H.: Concurrent Library Correctness on the TSO Memory Model. In: Seidl, H. (ed.) *ESOP 2012*. LNCS, vol. 7211, pp. 87–107. Springer, Heidelberg (2012)
6. Burstall, R.M.: Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence* 7, 23–50 (1972)
7. Coleman, J.W., Jones, C.B.: A structural proof of the soundness of rely/guarantee rules. *Journal of Logic and Computation* 17(4), 807–841 (2007)
8. Collette, P., Jones, C.B.: Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. In: Plotkin, G., Stirling, C., Tofte, M. (eds.) *Proof, Language and Interaction*, ch. 10, pp. 277–307. MIT Press (2000)
9. Cousot, P.: The Verification Grand Challenge and Abstract Interpretation. In: Meyer, B., Woodcock, J. (eds.) *VSTTE 2005*. LNCS, vol. 4171, pp. 189–201. Springer, Heidelberg (2008)
10. de Roeper, W.P.: *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press (2001)
11. Dijkstra, E.W.: *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs (1976)
12. Dingel, J.: A refinement calculus for shared-variable parallel and distributed programming. *Formal Aspects of Computing* 14, 123–197 (2002)
13. Jürgen Dingel. *Systematic Parallel Programming*. PhD thesis, Carnegie Mellon University, CMU-CS-99-172 (2000)

14. Dodds, M., Feng, X., Parkinson, M., Vafeiadis, V.: Deny-Guarantee Reasoning. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 363–377. Springer, Heidelberg (2009)
15. Feng, X.: Local rely-guarantee reasoning. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, pp. 315–327. ACM, New York (2009)
16. Floyd, R.W.: Assigning meanings to programs. In: Proc. Symp. in Applied Mathematics, vol. 19: Mathematical Aspects of Computer Science, pp. 19–32. American Mathematical Society (1967)
17. Gotsman, A., Yang, H.: Liveness-Preserving Atomicity Abstraction. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 453–465. Springer, Heidelberg (2011)
18. Hayes, I.J., Jones, C.B., Colvin, R.J.: Refining rely-guarantee thinking. Technical Report CS-TR-1334, Newcastle University (May 2012), submitted to Formal Aspects of Computing visible online, <http://www.cs.ncl.ac.uk/research/pubs/trs/papers/1334.pdf>
19. Hayes, I. (ed.): Specification Case Studies, 2nd edn. Prentice Hall International, Englewood Cliffs (1993)
20. Hayes, I.J., Burns, A., Dongol, B., Jones, C.B.: Comparing models of nondeterministic expression evaluation. Technical Report CS-TR-1273, School of Computing Science, University of Newcastle, Submitted to Computer Journal (September 2011) visible on-line, <http://www.cs.ncl.ac.uk/research/pubs/trs/papers/1273.pdf>
21. Hayes, I.J., Colvin, R.J.: Integrated Operational Semantics: Small-Step, Big-Step and Multi-step. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 21–35. Springer, Heidelberg (2012)
22. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
23. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580, 583 (1969)
24. Hoare, C.A.R., Hayes, I.J., He, J., Morgan, C., Roscoe, A.W., Sanders, J.W., Sørensen, I.H., Spivey, J.M., Sufrin, B.A.: The laws of programming. Communications of the ACM 30, 672–687 (1987), see Corrigenda in *ibid* 30:770
25. Hoare, C.A.R.: Towards a theory of parallel programming. In: Operating System Techniques, pp. 61–71. Academic Press (1972)
26. Isthiaq, S., O’Hearn, P.W.: BI as an assertion language for mutable data structures. In: 28th POPL, pp. 36–49 (2001)
27. Jones, C.B.: Software Development: A Rigorous Approach. Prentice Hall International, Englewood Cliffs (1980)
28. Jones, C.B.: Development Methods for Computer Programs including a Notion of Interference. PhD thesis, Oxford University, Printed as: Programming Research Group, Technical Monograph 25 (June 1981)
29. Jones, C.B.: Specification and design of (parallel) programs. In: Proceedings of IFIP 1983, pp. 321–332. North-Holland (1983)
30. Jones, C.B.: Tentative steps toward a development method for interfering programs. Transactions on Programming Languages and System 5(4), 596–619 (1983)
31. Jones, C.B.: Accommodating interference in the formal design of concurrent object-based programs. Formal Methods in System Design 8(2), 105–122 (1996)
32. Jones, C.B.: Wanted: a compositional approach to concurrency. In: McIver, A., Morgan, C. (eds.) Programming Methodology, pp. 5–15. Springer (2003)
33. Jones, C.B.: Splitting atoms safely. Theoretical Computer Science 375(1-3), 109–119 (2007)

34. Jones, C.B.: The role of auxiliary variables in the formal development of concurrent programs. In: Jones, C.B., Roscoe, A.W., Wood, K. (eds.) *Reflections on the Work of C.A.R. Hoare*, ch. 8, pp. 167–188. Springer (2010)
35. Jones, C.B.: The early search for tractable ways of reasoning about programs. *IEEE, Annals of the History of Computing* 25(2), 26–49 (2003)
36. Jones, C.B., Hayes, I.J., Jackson, M.A.: Deriving Specifications for Systems That Are Connected to the Physical World. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) *Bjørner/Zhou Festschrift*. LNCS, vol. 4700, pp. 364–390. Springer, Heidelberg (2007)
37. Jones, C.B., Pierce, K.G.: Splitting Atoms with Rely/Guarantee Conditions Coupled with Data Reification. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) *ABZ 2008*. LNCS, vol. 5238, pp. 360–377. Springer, Heidelberg (2008)
38. Jones, C.B., Pierce, K.G.: Elucidating concurrent algorithms via layers of abstraction and reification. *Formal Aspects of Computing* 23(3), 289–306 (2011)
39. Leinenbach, D., Santen, T.: Verifying the Microsoft Hyper-V Hypervisor with VCC. In: Cavalcanti, A., Dams, D.R. (eds.) *FM 2009*. LNCS, vol. 5850, pp. 806–809. Springer, Heidelberg (2009)
40. Liang, H., Feng, X., Fu, M.: A rely-guarantee-based simulation for verifying concurrent program transformations. In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012*, pp. 455–468. ACM, New York (2012)
41. Morgan, C.C.: *Programming from Specifications*, 2nd edn. Prentice Hall (1994)
42. Morgan, C.C., Vickers, T.N.: Types and invariants in the refinement calculus. In: Morgan, C.C., Vickers, T.N. (eds.) *On the Refinement Calculus*, pp. 127–154. Springer (1994)
43. Nipkow, T.: Non-deterministic data types: Models and implementations. *Acta Informatica* 22, 629–661 (1986)
44. O’Hearn, P.W., Reynolds, J.C., Yang, H.: Local Reasoning about Programs that Alter Data Structures. In: Fribourg, L. (ed.) *CSL 2001 and EACSL 2001*. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001)
45. O’Hearn, P.W.: Resources, concurrency and local reasoning. *Theoretical Computer Science* 375(1-3), 271–307 (2007)
46. O’Hearn, P.W., Pym, D.J.: The logic of bunched implications. *Bulletin of Symbolic Logic* 5(2), 215–244 (1999)
47. O’Hearn, P.W., Yang, H., Reynolds, J.C.: Separation and information hiding. *ACM TOPLAS* 31(3) (April 2009); Preliminary version appeared in 31st POPL, pp. 268–280 (2004)
48. Parkinson, M., Bornat, R., Calcagno, C.: Variables as resource in Hoare logics. In: *2006 21st Annual IEEE Symposium on Logic in Computer Science*, pp. 137–146 (2006)
49. Parkinson, M.: The Next 700 Separation Logics. In: Leavens, G.T., O’Hearn, P., Rajamani, S.K. (eds.) *VSTTE 2010*. LNCS, vol. 6217, pp. 169–182. Springer, Heidelberg (2010)
50. Parkinson, M., Bierman, G.: Separation logic and abstraction. In: *POPL 2005: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 247–258. ACM, New York (2005)
51. Reynolds, J.C.: Intuitionistic reasoning about shared mutable data structure. In: Davies, J., Roscoe, B., Woodcock, J. (eds.) *Millennial Perspectives in Computer Science*, Houndsmill, Hampshire, pp. 303–321. Palgrave (2000)
52. Reynolds, J.: A logic for shared mutable data structures. In: Plotkin, G. (ed.) *Proceedings of the Seventeenth Annual IEEE Symp. on Logic in Computer Science, LICS 2002*. IEEE Computer Society Press (July 2002)
53. Sangiorgi, D.: Typed π -calculus at work: a correctness proof of Jones’s parallelisation transformation on concurrent objects. *Theory and Practice of Object Systems* 5(1), 25–34 (1999)

54. Simpson, H.R.: Four-slot fully asynchronous communication mechanism. *IEE Proceedings E Computers and Digital Techniques* 137(1), 17–30 (1990)
55. Stølen, K.: *Development of Parallel Programs on Shared Data-Structures*. PhD thesis, Manchester University, Available as UMCS-91-1-1 (1990)
56. Vafeiadis, V.: *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge (2007)
57. Vafeiadis, V., Parkinson, M.: A Marriage of Rely/Guarantee and Separation Logic. In: Caires, L., Vasconcelos, V.T. (eds.) *CONCUR 2007*. LNCS, vol. 4703, pp. 256–271. Springer, Heidelberg (2007)

A Rule-Based and Imperative Language for Biochemical Modeling and Simulation

Đurica Nikolić^{1,2}, Corrado Priami^{1,3}, and Roberto Zunino^{1,3}

¹ The Microsoft Research - University of Trento Centre for Computational and Systems Biology

² University of Verona, Italy

³ University of Trento, Italy

Abstract. We present COSBI LAB Language (\mathcal{L} for short), a simple modeling language for biochemical systems. \mathcal{L} features stochastic multiset rewriting, defined in part through rewriting rules, and in part through imperative code.

We provide a continuous-time Markov chain semantics for \mathcal{L} at three different abstraction levels, linked by Galois connections. We then describe a simulation algorithm for the most concrete semantics, which is then adapted to work at higher abstract levels while improving space and time performance. Doing so results in the well-known Gillespie's Direct Method, as well as in a further optimized algorithm.

1 Introduction

In this paper we present a computer language, called \mathcal{L} , for modeling and simulating biochemical systems. In such setting, we are concerned with the modeling of the kinds of behaviour leading to the creation of *bimolecular complexes* and their mutual interaction. Complexes are to be thought as an aggregation of smaller molecules, kept together by chemical bonds on specific zones called *interaction sites*.

Different kinds of mathematical structures have been used to model such entities. Often, these take inspiration from *graphs* and their generalizations, e.g., hypergraphs. Here, the smaller molecules are represented by graph nodes, which are taken as primitive stateful entities. We shall name these stateful nodes “*boxes*”. Boxes also have a list of sites, from which they can be connected to other boxes by (undirected) edges. In this framework, complexes are just the connected components of the graphs. System evolution is then typically modeled via a stochastic transition system, hence providing a semantics based on continuous-time Markov chains; this is done to capture the inherent uncertainty of the biological phenomena, which are “noisy” in their nature. The actual definition of the transition system depends on the modeling language at hand. For instance, BlenX [5] is a language which uses a graph-like representation of complexes, whose boxes are equipped with a process in a *stochastic process algebra*. There, the stochastic operational semantics of the processes inside the boxes form the basis for defining the transition system for graphs. Kappa [2] instead resorts to stochastic *rewriting rules*, borrowing from graph rewriting techniques. These rules can be used to express in an intuitive way how the graph is affected by biochemical reactions.

Often, however, the precise graph structure of complexes, their chemical bonds, and the nature of the interaction sites is still unknown to researchers in biology. Modelers

wishing to use graph-based modeling languages are then asked to provide more data than those available. In such cases, it would be better to use a less detailed structure to represent complexes. In our \mathcal{L} language we use box *multisets* in lieu of box graphs. Consequently, interaction sites are no longer represented, as well as exact chemical bonds. Rather, we just represent the fact that two molecules belong to the same complexes (or to different ones). This approach seem to be closer to the actual knowledge available to researchers. As a bonus, we also get some performance improvements in simulation, since e.g. graph isomorphism tests (untractable, but quadratic time under certain assumptions [7]) are now replaced by multiset equality tests (linear time with most representations).

In \mathcal{L} , we use stochastic multiset rewriting rules to define the evolution of a system. Three kind of rules are used: association rules (*assoc*) which merge two complexes, dissociation rules (*disso*) which split them, and general dynamics *dyn* to model the rest of the interactions. These rules are defined using complex *patterns*, selecting which complexes are to be rewritten. Rewriting is, in part, automatic (for *assoc* and *disso*) and can be augmented by *imperative code* whenever one needs to describe custom multiset manipulations. The effect of this code is atomic: no other rule firing is interleaved. Having atomic “large” effects is useful, since otherwise such effects need to be carefully programmed across many rules, often using “infinite” stochastic rates and dealing with all the resulting concurrency issues [6]. For instance, if we want to model a vesicle releasing at once all its carried molecules (the number of which is not statically known), we can not use a basic rule, yet it is straightforward to program this behaviour.

We present the syntax of \mathcal{L} in Sect. 2, and its semantics in Sect. 3. We then study how the language can be efficiently simulated. In order to do so, we apply *abstract interpretation* and construct two more abstract semantics in Sect. 4. We then apply this to construct efficient simulation algorithms for \mathcal{L} in Sect. 5. We start from a simple, yet inefficient algorithm which is then adapted to exploit the abstract semantics. This results in improvements to space and time performance. The algorithms constructed in this way are the well-known Gillespie’s Direct Method [4] as well as a new improvement of it which better takes advantage of the features of \mathcal{L} .

2 Syntax

In Figure 1 we show the syntax of the \mathcal{L} language. We give a short comment on the constructs in that figure. *BasicType* stands for a primitive type used in our language, while *BasicLiteral* ranges over their values. The modeler can declare a *box* type, by specifying a name for it and of a set of *fields* having basic types. More formally, *FieldDecl* is a sequence of field declarations of form *Field* : *BasicType*, where *Field* is the name of the field, and is unique inside the box. The field list can be empty (ϵ). Then, the declaration of a box type *BoxDecl* has the form *BoxType*{*FieldDecl*}, where *BoxType* is a name for the declared type. Moreover, *FieldInit* is a (possibly empty) sequence of initialized fields, while *BoxLiteral* represents a box having all its declared fields instantiated. For example, $A\{x : \text{int}; y : \text{real}\}$ is a declaration of a box type *A* containing fields *x* and *y* of the given types, and $A\{x = 3; y = 1.0\}$ is an instantiation. We use **Box** to denote the set of all possible box instantiations.

| | |
|------------------------|--|
| <i>BasicType</i> | ::= bool int real |
| <i>BasicLiteral</i> | ::= <i>BoolLiteral</i> <i>IntLiteral</i> <i>RealLiteral</i> |
| <i>Field</i> | ::= <i>Ide</i> |
| <i>FieldDecl</i> | ::= ϵ <i>Field</i> : <i>BasicType</i> ; <i>FieldDecl</i> |
| <i>FieldInit</i> | ::= ϵ <i>Field</i> = <i>BasicLiteral</i> ; <i>FieldInit</i> |
| <i>Exp</i> | ::= <i>BasicLiteral</i> null <i>Ide</i> <i>Exp</i> \wedge <i>Exp</i> \neg <i>Exp</i> <i>Exp</i> + <i>Exp</i> <i>Exp</i> - <i>Exp</i> <i>Exp</i> * <i>Exp</i> <i>Exp</i> = <i>Exp</i> <i>Exp</i> < <i>Exp</i> <i>Exp</i> . <i>Field</i> <i>Exp</i> .count(<i>BoxType</i>) <i>Exp</i> .first(<i>BoxType</i>) |
| <i>BoxType</i> | ::= <i>Ide</i> |
| <i>BoxDecl</i> | ::= <i>BoxType</i> { <i>FieldDecl</i> } |
| <i>BoxLiteral</i> | ::= <i>BoxType</i> { <i>FieldInit</i> } NOTE: all declared fields must be instantiated |
| <i>CplxLiteral</i> | ::= [<i>IntLiteral</i> : <i>BoxLiteral</i> ; <i>CplxLiteralTail</i>] |
| <i>CplxLiteralTail</i> | ::= ϵ <i>IntLiteral</i> : <i>BoxLiteral</i> ; <i>CplxLiteralTail</i> |
| <i>Complexes</i> | ::= <i>IntLiteral</i> : <i>CplxLiteral</i> ; <i>ComplexesTail</i> |
| <i>ComplexesTail</i> | ::= ϵ <i>IntLiteral</i> : <i>CplxLiteral</i> ; <i>ComplexesTail</i> |
| <i>Pattern</i> | ::= [<i>BoxType</i> { <i>FieldInit</i> } <i>PatternTail</i>] |
| <i>PatternTail</i> | ::= ϵ , <i>BasePattern</i> |
| <i>Assoc</i> | ::= assoc <i>Pattern</i> <i>Pattern</i> rate <i>Exp</i> react <i>Block</i> |
| <i>Dissoc</i> | ::= dissoc <i>Pattern</i> <i>Pattern</i> ^{no*} rate <i>Exp</i> react <i>Block</i> |
| <i>Dyn</i> | ::= dyn <i>Pattern</i> ₁ . . . <i>Pattern</i> _n rate <i>Exp</i> react <i>Block</i> |
| <i>Block</i> | ::= var <i>Ide</i> := <i>Exp</i> ; <i>Block</i> <i>CmdBlock</i> |
| <i>CmdBlock</i> | ::= end <i>Cmd</i> ; <i>CmdBlock</i> |
| <i>Cmd</i> | ::= skip <i>Ide</i> := <i>Exp</i> if <i>Exp</i> then <i>Block</i> else <i>Block</i> while <i>Exp</i> do <i>Block</i> <i>BoxCommand</i> |
| <i>BoxCommand</i> | ::= <i>Ide</i> := <i>Exp</i> .spawn(<i>BoxLiteral</i> , . . .) <i>Exp</i> .spawn(<i>Exp</i>) <i>Exp</i> .remove(<i>Exp</i>) <i>Exp</i> .merge(<i>Exp</i>) <i>Exp</i> .move(<i>Exp</i> , <i>Exp</i>) foreach <i>Ide</i> : <i>BoxType</i> in <i>Exp</i> do <i>Block</i> <i>Exp</i> . <i>Ide</i> := <i>Exp</i> |
| <i>Decl</i> | ::= <i>Assoc</i> <i>Dissoc</i> <i>Dyn</i> <i>BoxDecl</i> |
| <i>Run</i> | ::= run <i>Complexes</i> end |
| <i>Model</i> | ::= <i>Run</i> <i>Decl</i> ; <i>Model</i> |

Fig. 1. Syntax of the \mathcal{L} language

For any set S , we write $\text{mset } S$ for the set of multisets over S , which we sometimes identify with the set of functions $S \rightarrow \mathbb{N}$. A *complex* is a multiset of boxes, which we represent in our syntax by a *CplxLiteral*. The latter is a non-empty sequence of the form *IntLiteral* : *BoxLiteral*, where *IntLiteral* denotes how many instances of *BoxLiteral* are present in the complex. When *IntLiteral* = 1, we omit to write it. For instance, $[2 : A\{x = 3; y = 1.0\}, B\{\}]$ is a complex. We use $\text{Cplx} = \text{mset } \text{Box}$ to denote the set of all possible complexes. The whole system state is then defined via the *Run* clause, which specifies an initial sequence of *Complexes*, having form *IntLiteral* : *CplxLiteral* where *IntLiteral* represents the initial population of the complex in the system at hand.

The dynamics of the system is given by multiset rewriting rules, which continuously modify the system at hand (if no rule applies, the system does not evolve further). Our rules are based on complex patterns. A *Pattern* is a sequence of literals of form *BoxType*{*FieldInit*}, possibly followed by a wildcard *. Intuitively, a pattern without * matches with complexes having exactly the specified boxes, while the wildcard allows matching with complexes including other boxes as well. More formally, we say that a box $B_1\{f_1 = v_1, \dots, f_n = v_n\}$ matches with a box $B_2\{g_1 = h_1, \dots, g_m = h_m\}$ if $B_1 = B_2$,

$n \leq m$ and for each $i \in [1..n]$ there exists $j \in [1..m]$ such that $f_i = g_j$ and $v_i = h_j$. Then, we say that a complex $c \in \text{Cplx}$ matches with a pattern p , denoted with $c \models p$, if one of the following conditions holds:

- p does not end with $*$, and there is a *bijjective* correspondence between boxes in p and those in c , where correspondent boxes match.
- p does end with $*$, and there is an *injective* correspondence between boxes in p and those in c , where correspondent boxes match.

The following example illustrates some pattern matchings.

Example 1. Consider complexes $c_1 = [A\{x = 1\}]$, $c_2 = [B\{\}]$, $c_3 = [A\{x = 0\}, A\{x = 1\}]$, $c_4 = [A\{x = 1\}, B\{\}]$ and $c_5 = [A\{x = 1, y = 4\}]$ and patterns $p_1 = [A]$, $p_2 = [A, A]$, $p_3 = [A, *]$, $p_4 = [B]$, $p_5 = [B, *]$ and $p_6 = [A\{x = 1\}]$. Then, only the following relations hold: $c_1 \models p_1$, $c_1 \models p_3$, $c_1 \models p_6$, $c_2 \models p_4$, $c_2 \models p_5$, $c_3 \models p_2$, $c_3 \models p_3$, $c_4 \models p_3$, $c_4 \models p_5$, $c_5 \models p_1$, $c_5 \models p_3$, $c_5 \models p_6$. ■

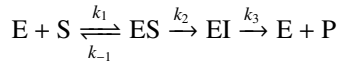
Having defined patterns, we can now discuss the rewriting rules which lead the evolution of the system. Our language features three kinds of such rules, namely *Assoc*, *Dissoc* and *Dyn*. A rule `assoc p_1 p_2 rate Exp react $Block$` allows pairs of complexes matching with p_1 and p_2 to associate. When that happens, the two reactant complexes merge their boxes and form a new larger complex, mimicking the association of two macromolecules in biological systems. The rate expression Exp provides the rate for the stochastic transition, thus defining the “speed” of the association. The rate expression is allowed to inspect the boxes in the two reactants via two special variables `reactant1` and `reactant2`. For instance, `assoc [A] [B, *] rate 5.2 * reactant1.first(A).mass` defines a rate proportional to the mass of the first reactant. When an `assoc` rule is fired, after the complexes are associated the code block specified in the `react` part is run. This can access the newly formed product (via a special `product` variable) and modify it further, e.g. by changing box fields, or adding/removing boxes. The `react` code block can also spawn entirely new complexes.

A rule `dissoc p_1 p_2 rate Exp react $Block$` specifies the dual operation, namely dissociation of a complex into two subcomplexes. Here, p_1 specifies the complex to break up, while p_2 matches with a subcomplex to separate (no wildcard $*$ is allowed in p_2). The rate expression Exp can access `reactant1` to provide a dissociation stochastic rate, which is intended to define how fast is the reactant to split. In the case p_2 has multiple matches inside the reactant, we let all of them define an equally probable dissociation, hence effectively dividing the rate among all the possible splits. After the rule triggers and the split is performed, the `react` code block is run, and can access the new complexes using the variables `product1` and `product2`.

Rule `dyn` is used to define a generic molecular dynamics. Its semantics is as for `assoc`, except that no complex merge is performed, and the `react` code block still has access to the unmerged complexes `reactant1` and `reactant2`. This rule effectively subsumes `assoc` and `dissoc`, in that association/dissociation can be programmed manually in the code block. However, associations and dissociations are so common to deserve a special construct. Instead, the typical use for `dyn` is the modeling of monomolecular reactions, in which only one reactant is present.

The code blocks in rules are written in an imperative language, the constructs of which are mostly standard. Therefore, we just briefly discuss the more peculiar ones. Since the state of \mathcal{L} is stored in complexes, i.e. in multisets of boxes, we need constructs to inspect and modify those. We provide a way to loop over all the boxes of a given type in a complex (`foreach $b : BoxType$ in $complex$`). To precisely define the semantics of such loop, we require that the visit order follows the lexicographic order of box values. The expression `$complex.first(BoxType)$` returns the first box in such ordering. Further commands allows one to add (`$complex.spawn$`) and remove (`$complex.remove$`) boxes in a complex. Similar operations can be done at the complex level: new complexes can be created, and existing ones removed. We provide also ways to move boxes between complexes (`$move$`) as well as to merge two complex as it happens for association (`$merge$`).

Example 2. As a simple example, we provide an \mathcal{L} model for the enzymatic reaction shown below:



The first double arrow models an enzyme molecule (E) associating and dissociating to a substrate molecule (S). When associated, the complex ES can react (second arrow): the enzyme changes the substrate into some intermediate molecule (I). This reaction is not reversible. Then, the intermediate molecule can dissociate from the enzyme, which releases a product (P) in the system (third arrow). In \mathcal{L} , we can model this behaviour as follows. Below, the react blocks are used to change S into I , and then I into P .

```

E{}   S{}   I{}   P{}
assoc [E] [S] rate k1;
dissoc [E, S] [S] rate k-1;
dyn [E, S] rate k2 react
    reactant1.remove(reactant1.first(S));
    reactant1.spawn(I{});
    end;
dissoc [E, I] [I] rate k3 react
    product2.remove(product2.first(I));
    product2.spawn(P{});
    end;
run 100 : [E]; 100 : [S]; end

```

■

3 Semantics

In this section we provide a semantics for the rules of our \mathcal{L} language. To keep our presentation short, we focus on the semantics of the `assoc` rule, only. The formal semantics of the `dissoc` and `dyn` rules can be indeed defined similarly.

Suppose we are given a set of complexes annotated with their *names*. Names uniquely identify the instances of complexes present in the system, so that complexes comprised by exactly the same boxes are still distinguishable. For the sake of simplicity, we assume that complexes' names are natural numbers. Then, suppose that we are

given an `assoc` rule which states that all the pairs of complexes satisfying patterns p_1 and p_2 react with a stochastic rate specified by an expression e . The semantics of the `assoc` rule then collects the stochastic transitions involving all such pairs of complexes, and their corresponding rates.

We start by defining two auxiliary binary set operators, which are similar to the cartesian product. Operator \otimes defines the set of *unordered* pairs $(\{x_1, x_2\})$, while operator $\hat{\otimes}$ defines the set of *unordered* pairs of *distinct* elements (hence requiring $x_1 \neq x_2$).

Definition 1 (\otimes and $\hat{\otimes}$). *Given two arbitrary sets S_1 and S_2 , we define*

$$S_1 \otimes S_2 = \{\{x_1, x_2\} \mid x_1 \in S_1 \wedge x_2 \in S_2\} \quad S_1 \hat{\otimes} S_2 = \{\{x_1, x_2\} \mid x_1 \in S_1 \wedge x_2 \in S_2 \wedge x_1 \neq x_2\}.$$

The following definition introduces the notion of system, representing a set of complexes enriched with their names, and the notion of transition, representing two different complexes that can react with a certain rate.

Definition 2 (Systems and Transitions). *A system σ^b is a set of annotated complexes, i.e., ordered pairs $\langle n, c \rangle \in \mathbb{N} \times \text{Cplx}$, where c and n represent a complex and its unique name, respectively. We write $\text{Sys}_0 = \wp(\mathbb{N} \times \text{Cplx})$ to denote the set of all the possible systems. A transition is an ordered pair whose first element determines the reactants, while the second element is the rate of the reaction. The reactants are characterized by an unordered pair of annotated complexes. We write $\text{Tr}_0 = ((\mathbb{N} \times \text{Cplx}) \hat{\otimes} (\mathbb{N} \times \text{Cplx})) \times \mathbb{R}^+$ to denote the set of all possible transitions. Moreover, we let $\text{Res}_0 = \wp(\text{Tr}_0)$.*

It is worth noting that Res_0 denotes all possible sets of transitions, and therefore represents the set of all possible semantics of an `assoc` rule.

Example 3. Let $c_1 = [A\{x = 0\}]$, $c_2 = [A\{x = 1\}]$ and $c_3 = [B]$ be three complexes, and suppose that a system σ^b is composed of 2, 1 and 2 instances of c_1 , c_2 and c_3 respectively. Then, we represent σ^b as follows: $\sigma^b = \{\langle 0, c_1 \rangle, \langle 1, c_1 \rangle, \langle 0, c_2 \rangle, \langle 0, c_3 \rangle, \langle 1, c_3 \rangle\}$. If we suppose that $\langle 0, c_1 \rangle$ and $\langle 0, c_2 \rangle$ can react with a rate 1.0, the corresponding transition is given as: $\{\langle \langle 0, c_1 \rangle, \langle 0, c_2 \rangle \rangle, 1.0\}$. ■

We now define a function which selects from a system only those annotated complexes which match with a given pattern.

Definition 3. *Given a pattern p , we define a map $\llbracket p \rrbracket^0 : \text{Sys}_0 \rightarrow \text{Sys}_0$ called the evaluation of p in Sys_0 as: $\llbracket p \rrbracket^0 \sigma^b = \{\langle n, c \rangle \in \sigma^b \mid c \models p\}$, for any $\sigma^b \in \text{Sys}_0$.*

Example 4. Consider the system σ^b defined in Example 3 and a pattern $[A]$ denoting the complexes composed of only one box whose its type is A . Since complexes c_1 and c_2 match with $[A]$, while c_3 does not, we have $\llbracket [A] \rrbracket^0 \sigma^b = \{\langle 0, c_1 \rangle, \langle 1, c_1 \rangle, \langle 0, c_2 \rangle\}$. ■

We now define to the actual semantics of the `assoc` rule. Its general form is the following: `assoc p_1 p_2 rate e react Block`. Starting from a system σ^b , we evaluate patterns p_1 and p_2 (Definition 3), obtaining two systems σ_1^b and σ_2^b whose complexes match with patterns p_1 and p_2 respectively. Our goal is to determine all possible transitions whose first reactant belongs to σ_1^b , and the second one belongs to σ_2^b . The rate of each

transition is determined by evaluating the expression e . This evaluation depends on an *environment*, i.e., a function which maps special variables reactant_1 and reactant_2 to their values. These values are the complexes which take part in the transition. The following definition explains how it is possible to evaluate the rate characterized by an expression e of a reaction between two complexes c_1 and c_2 .

It is worth noting that the rate expression e should be “symmetric” in reactant_1 and reactant_2 : swapping their values should not affect the resulting association rate. This reflects the fact that in biology association happens between unordered complex pairs. To stress and enforce this symmetry, below we compute the rate as the average of the rates resulting from both orders.

Definition 4. Let c_1 and c_2 be two complexes and let e be an expression characterizing the rate of the reaction between these complexes. We determine the rate of this reaction as follows:

$$\text{rate}(c_1, c_2, e) = \frac{\llbracket e \rrbracket^{\text{exp}} \rho_1 + \llbracket e \rrbracket^{\text{exp}} \rho_2}{2},$$

where $\rho_1 = [\text{reactant}_1 \mapsto c_1, \text{reactant}_2 \mapsto c_2]$ and $\rho_2 = [\text{reactant}_1 \mapsto c_2, \text{reactant}_2 \mapsto c_1]$.

Given an `assoc` rule and a system, we define the set of all possible transitions that can occur between complexes of the system.

Definition 5 (Semantics of `assoc`). Consider a rule `assoc` p_1 p_2 rate e react B and a state $\sigma^b \in \text{Sys}_0$. We define the map $\llbracket p_1, p_2, e \rrbracket_0^{\text{assoc-nc}} : \text{Sys}_0 \rightarrow \text{Res}_0$ as

$$\llbracket p_1, p_2, e \rrbracket_0^{\text{assoc-nc}} \sigma^b = \{ \langle \langle n_1, c_1 \rangle, \langle n_2, c_2 \rangle \rangle, \text{rate}(c_1, c_2, e) \rangle \in \text{Tr}_0 \mid \langle n_1, c_1 \rangle \in \llbracket p_1 \rrbracket_0^0 \sigma^b \wedge \langle n_2, c_2 \rangle \in \llbracket p_2 \rrbracket_0^0 \sigma^b \}.$$

The collecting semantics $\llbracket p_1, p_2, e \rrbracket_0^{\text{assoc}} : \wp(\text{Sys}_0) \rightarrow \wp(\text{Res}_0)$ is defined as follows: $\llbracket p_1, p_2, e \rrbracket_0^{\text{assoc}} \Sigma^b = \{ \llbracket p_1, p_2, e \rrbracket_0^{\text{assoc-nc}} \sigma^b \mid \sigma^b \in \Sigma^b \}$.

Note that the semantics of the `assoc` rule contains only transitions of type Tr_0 , i.e., the ones in which reactants must be different. This does not prevent two identical complexes to react, since they have different names. However, a complex is prevented to react with itself, which would be unwanted.

Example 5. Let us consider, one more time, the system σ^b defined in Example 3 and let us determine the semantics of the rules

$$\text{rule}_1 : \text{assoc } [A] [B] \text{ rate } 1.0 \quad \text{rule}_2 : \text{assoc } [A] [A] \text{ rate } 1.0.$$

In Example 4 we showed that $\llbracket [A] \rrbracket_0^0 \sigma^b = \{ \langle 0, c_1 \rangle, \langle 1, c_1 \rangle, \langle 0, c_2 \rangle \}$. We can, similarly show that $\llbracket [B] \rrbracket_0^0 \sigma^b = \{ \langle 0, c_3 \rangle, \langle 1, c_3 \rangle \}$. The following table represents the semantics of the rules: rule_1 gives rise to transitions 1.-6. from the following table, while rule_2 gives rise to transitions 7.-9.

| | | | | | | |
|-------------------|----|---|----|---|----|---|
| rule ₁ | 1. | $\langle \langle 0, c_1 \rangle, \langle 0, c_3 \rangle \rangle, 1$ | 2. | $\langle \langle 0, c_1 \rangle, \langle 1, c_3 \rangle \rangle, 1$ | 3. | $\langle \langle 1, c_1 \rangle, \langle 0, c_3 \rangle \rangle, 1$ |
| | 4. | $\langle \langle 1, c_1 \rangle, \langle 1, c_3 \rangle \rangle, 1$ | 5. | $\langle \langle 0, c_2 \rangle, \langle 0, c_3 \rangle \rangle, 1$ | 6. | $\langle \langle 0, c_2 \rangle, \langle 1, c_3 \rangle \rangle, 1$ |
| rule ₂ | 7. | $\langle \langle 0, c_1 \rangle, \langle 0, c_2 \rangle \rangle, 1$ | 8. | $\langle \langle 1, c_1 \rangle, \langle 0, c_2 \rangle \rangle, 1$ | 9. | $\langle \langle 0, c_1 \rangle, \langle 1, c_1 \rangle \rangle, 1$ |

Transitions 1.-6. occur between a complex matching with $[A]$ and a complex matching with $[B]$ and their rate is 1, while in transitions 7.-9. both complexes match with $[A]$. ■

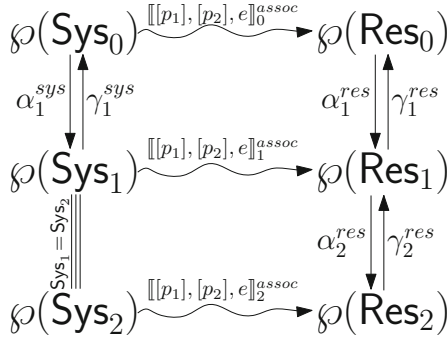


Fig. 2. Relationships between different levels of abstraction

Below, we state a simple property of the semantics of our expressions and commands. As it happens in nominal calculi and in most languages with references, we have that the actual names (n) used to annotate the “objects” (the complexes $\langle n, c \rangle$) are immaterial.

Lemma 1. *The semantics of expressions, commands, and rewriting rules are stable w.r.t. the renaming of annotated complexes. In particular, renaming does not affect the computed transition rates for rules, or their modifications of the state.*

4 Abstraction

There might be a huge number of annotated complexes satisfying the two patterns specified by an `assoc` rule that react. Hence, the number of reactions they give rise can become enormous. Although precise, the provided semantics for rules can lead to an inefficient simulation, as we will see. In this section we define two levels of abstraction which we shall exploit to perform some optimization in the simulation of \mathcal{L} models. We call the actual semantics of the `assoc` rules defined in Section 3 the semantics of *level 0*, while we refer to our new abstraction levels as *level 1* and *level 2*. For both of them, we define some opportune abstractions of Sys_0 (Sys_1 and Sys_2) and Res_0 (Res_1 and Res_2), show how they can be related (via pairs of functions $\alpha^{\text{sys}}-\gamma^{\text{sys}}$ and $\alpha^{\text{res}}-\gamma^{\text{res}}$) and define two abstract semantics ($\llbracket p_1, p_2, e \rrbracket_1^{\text{assoc}}$ and $\llbracket p_1, p_2, e \rrbracket_2^{\text{assoc}}$) representing a sound approximation of the actual semantics $\llbracket p_1, p_2, e \rrbracket_0^{\text{assoc}}$ of the `assoc` rules. In Figure 2 we give a brief outline of this idea which we then formalize in Sections 4.1 and 4.2.

4.1 Abstraction at Level 1

In Section 3 we have stated that the information about complexes’ names is not relevant for the actual semantics of the `assoc` rules. Therefore, it is possible to abstract away this piece of information and define another, more abstract semantics of these rules. In the following we will define counterparts of Definitions 2, 3 and 5.

Although the names of complexes are not relevant for the computation of reaction rates, it is important to know the exact amount of each complex present in a system,

i.e. its population. This fact gives rise to another definition of the notion of system, on a different level of abstraction. More precisely, for each complex, we memorize the number of instances of that complex present in a system. Then we remove information about complexes' names from the definition of transitions as well. The following definition formalizes these intuitions.

Definition 6. A system σ is a multiset of complexes, i.e., $\sigma \in \text{mset Cplx} = \text{Sys}_1$. A transition is an ordered pair whose first element determines the set of reactants, while the second element is the rate of the reaction. The set of reactants can be composed of 1 or 2 elements. The former case occurs when two identical complexes react, while the latter case occurs when the reactants are not identical. We write $\text{Tr}_1 = (\text{Cplx} \otimes \text{Cplx}) \times \mathbb{R}^+$ to denote the set of all possible transitions. Moreover, we let $\text{Res}_1 = \wp(\text{Tr}_1)$.

Example 6. Under the hypotheses of Example 3 regarding the structure of complexes c_1, c_2 and c_3 and their populations which are, respectively 2, 1 and 2, we define the state $\sigma = [c_1 \mapsto 2, c_2 \mapsto 1, c_3 \mapsto 2]$. ■

Definition 7. We define a function $\text{names} : \text{Cplx} \times \text{Sys}_0 \rightarrow \wp(\mathbb{N})$ which for every complex $c \in \text{Cplx}$ and every system $\sigma^b \in \text{Sys}_0$ returns all possible names that c might have in σ^b as $\text{names}(c, \sigma^b) = \{n \in \mathbb{N} \mid \langle n, c \rangle \in \sigma^b\}$.

It is worth noting that $\langle \wp(\text{Sys}_0), \subseteq, \cup, \cap, \emptyset, \text{Sys}_0 \rangle$, and $\langle \wp(\text{Sys}_1), \subseteq, \cup, \cap, \emptyset, \text{Sys}_1 \rangle$ (or shortly, $\langle \wp(\text{Sys}_0), \subseteq \rangle$ and $\langle \wp(\text{Sys}_1), \subseteq \rangle$) are complete lattices. In the following we show that they are also related by a Galois connection [11].

Definition 8. Let $\beta_1^{\text{sys}} : \text{Sys}_0 \rightarrow \text{Sys}_1$ be defined as $\beta_1^{\text{sys}}(\sigma^b) = \lambda c. |\text{names}(c, \sigma^b)|$. We define the abstraction map $\alpha_1^{\text{sys}} : \wp(\text{Sys}_0) \rightarrow \wp(\text{Sys}_1)$ and the concretization map $\gamma_1^{\text{sys}} : \wp(\text{Sys}_1) \rightarrow \wp(\text{Sys}_0)$ as:

$$\alpha_1^{\text{sys}}(\Sigma^b) = \{\beta_1^{\text{sys}}(\sigma^b) \mid \sigma^b \in \Sigma^b\} \quad \gamma_1^{\text{sys}}(\Sigma) = \bigcup \{\Sigma^b \mid \alpha_1^{\text{sys}}(\Sigma^b) \subseteq \Sigma\}.$$

Lemma 2. $\langle \langle \wp(\text{Sys}_0), \subseteq \rangle, \alpha_1^{\text{sys}}, \gamma_1^{\text{sys}}, \langle \wp(\text{Sys}_1), \subseteq \rangle \rangle$ is a Galois connection.

The abstraction map α_1^{sys} modifies a system σ^b by substituting the names of the complexes belonging to that system with their population. The definition of γ_1^{sys} depends on α_1^{sys} and derives from a well-known result from the theory of abstract interpretation. Its meaning is clarified by the following lemma.

Lemma 3. Let $\delta_1^{\text{sys}} : \text{Sys}_1 \rightarrow \wp(\text{Sys}_0)$ be a function defined as:

$$\delta_1^{\text{sys}}(\sigma) = \{\sigma^b \in \text{Sys}_0 \mid \forall c \in \text{Cplx}. |\text{names}(c, \sigma^b)| = \sigma(c)\}.$$

Then, $\gamma_1^{\text{sys}}(\Sigma) = \bigcup_{\sigma \in \Sigma} \delta_1^{\text{sys}}(\sigma)$.

In the following we perform a similar abstraction to the set of transitions Res_0 by removing all pieces of information regarding the names of the complexes that can react. At this point, it might be the case that more than one transition has the same reactants. We group all these transitions together and we assign them a rate obtained as sum of

the rates of each single transition. For instance, consider the transitions 5. and 6. from Example 5: $\langle\langle 0, c_2 \rangle, \langle 0, c_3 \rangle\rangle, 1, \langle\langle 0, c_2 \rangle, \langle 1, c_3 \rangle\rangle, 1 \in \text{Tr}_0$. By removing complexes' names, we have two transitions of the same form: $\langle\{c_2, c_3\}, 1\rangle$. The idea is then to substitute them with only one transition with rate $1 + 1 = 2$: $\langle\{c_2, c_3\}, 2\rangle \in \text{Tr}_1$. This rate is called *propensity*. We formalize our idea: the map *comp* we define below substitutes the reactants of each transition by the set composed of their complexes, while the map *prop* calculates the propensities of new transitions in the way we hinted at above.

Definition 9. We define maps $\text{comp} : \text{Res}_0 \rightarrow \wp(\text{Cplx} \otimes \text{Cplx})$ and $\text{prop} : (\text{Res}_0 \times (\text{Cplx} \otimes \text{Cplx})) \rightarrow \mathbb{R}$ as:

$$\begin{aligned} \text{comp}(R^b) &= \{\{c_1, c_2\} \mid \exists n_1, n_2 \in \mathbb{N}. \exists r \in \mathbb{R}^+. \langle\langle n_1, c_1 \rangle, \langle n_2, c_2 \rangle\rangle, r \in R^b\} \\ \text{prop}(R^b, A) &= \sum_{\langle B, r \rangle \in R^b \text{ s.t. } \text{comp}(\langle B, r \rangle) = A} r \end{aligned}$$

Example 7. Let R^b be the set composed of the 9 transitions obtained in Example 5. Then, $\text{comp}(R^b) = \{\{c_1\}, \{c_1, c_2\}, \{c_1, c_3\}, \{c_2, c_3\}\}$. On the other hand, the propensities are $\text{prop}(R^b, \{c_1\}) = 1$, $\text{prop}(R^b, \{c_1, c_2\}) = \text{prop}(R^b, \{c_2, c_3\}) = 2$, $\text{prop}(R^b, \{c_1, c_3\}) = 4$ and $\text{prop}(R^b, A) = 0$ for all other $A \in \wp(\text{Cplx} \otimes \text{Cplx})$. ■

We can now define another pair of abstraction and concretization maps in order to relate Res_0 and Res_1 .

Definition 10. We define the abstraction map $\alpha_1^{\text{res}} : \wp(\text{Res}_0) \rightarrow \wp(\text{Res}_1)$ and the concretization map $\gamma_1^{\text{res}} : \wp(\text{Res}_1) \rightarrow \wp(\text{Res}_0)$ as:

$$\begin{aligned} \alpha_1^{\text{res}}(\mathcal{R}^b) &= \{\langle A, \text{prop}(R^b, A) \rangle \mid A \in \text{comp}(R^b)\} \mid R^b \in \mathcal{R}^b\} \\ \gamma_1^{\text{res}}(\mathcal{R}) &= \bigcup \{R^b \mid \alpha_1^{\text{res}}(\mathcal{R}^b) \subseteq \mathcal{R}\}. \end{aligned}$$

Lemma 4. $\langle\langle \wp(\text{Res}_0), \subseteq \rangle, \alpha_1^{\text{res}}, \gamma_1^{\text{res}}, \langle \wp(\text{Res}_1), \subseteq \rangle \rangle$ is a Galois connection.

We define a map which removes from a system $\sigma \in \text{Sys}_1$ corresponding to a multiset of complexes (Definition 6) all those complexes not matching with a given pattern.

Definition 11. Given a pattern p and a system $\sigma \in \text{Sys}_1$, the evaluation of p in σ is a map $\llbracket p \rrbracket^1 : \text{Sys}_1 \rightarrow \text{Sys}_1$ defined as:

$$\llbracket p \rrbracket^1 \sigma = \lambda c. \begin{cases} \sigma(c) & \text{if } c \models p \\ 0 & \text{otherwise.} \end{cases}$$

Example 8. Consider the system σ defined in Example 6 and a pattern $[A]$. Since c_1 and c_2 match with p , while c_3 does not, we have $\llbracket [A] \rrbracket^1 \sigma = [c_1 \mapsto 2, c_2 \mapsto 1, c_3 \mapsto 0]$. ■

Given a system $\sigma \in \text{Sys}_1$, two complexes in it that may react, and the rate expression specified by an *assoc* rule, the following definition specifies how the propensity of all possible transitions induced by σ which have these complexes as reactants is computed.

Definition 12. Let c_1 and c_2 be two complexes appearing in a system $\sigma \in \text{Sys}_1$ and let e be an expression characterizing the rate of the reaction between these complexes. We determine the propensity of this reaction as follows:

$$p_\sigma(c_1, c_2, e) = \frac{\sigma(c) \cdot (\sigma(c) - 1)}{2} \cdot \llbracket e \rrbracket \rho_1 \quad p_\sigma(c_1, c_2, e) = \sigma(c_1) \cdot \sigma(c_2) \cdot \frac{\llbracket e \rrbracket \rho_2 + \llbracket e \rrbracket \rho_3}{2},$$

where the first equation applies when $c_1 = c_2 = c$, otherwise the second applies; also, above we let $\rho_1 = [\text{reactant}_1 \mapsto c, \text{reactant}_2 \mapsto c]$, $\rho_2 = [\text{reactant}_1 \mapsto c_1, \text{reactant}_2 \mapsto c_2]$ and $\rho_3 = [\text{reactant}_1 \mapsto c_2, \text{reactant}_2 \mapsto c_1]$.

In the following we define $\llbracket p_1, p_2, e \rrbracket_1^{\text{assoc}}$, the abstract semantics at level 1 of the actual semantics of the assoc rules (Definition 5).

Definition 13 (Level 1 semantics of assoc). Consider a state $\sigma \in \text{Sys}_1$ and a rule assoc $p_1 p_2$ rate e react B . We define the map $\llbracket p_1, p_2, e \rrbracket_1^{\text{assoc-nc}} : \text{Sys}_1 \rightarrow \text{Res}_1$ as

$$\llbracket p_1, p_2, e \rrbracket_1^{\text{assoc-nc}} \sigma = \{ \langle \{c_1, c_2\}, p_\sigma(c_1, c_2, e) \rangle \in \text{Tr}_1 \mid \llbracket p_1 \rrbracket^1 \sigma(c_1) \neq 0 \wedge \llbracket p_2 \rrbracket^1 \sigma(c_2) \neq 0 \}.$$

The collecting semantics $\llbracket p_1, p_2, e \rrbracket_1^{\text{assoc}} : \wp(\text{Sys}_1) \rightarrow \wp(\text{Res}_1)$ is defined as follows: $\llbracket p_1, p_2, e \rrbracket_1^{\text{assoc}} \Sigma = \{ \llbracket p_1, p_2, e \rrbracket_1^{\text{assoc-nc}} \sigma \mid \sigma \in \Sigma \}$.

Example 9. Let us consider, one more time, rule₁ and rule₂ introduced in Example 5 and the system σ defined in Example 6. Let us determine the semantics at level 1 of these rules in σ . In Example 8 we showed that $\llbracket [A] \rrbracket^1 \sigma = [c_1 \mapsto 2, c_2 \mapsto 1, c_3 \mapsto 0]$. We can, similarly show that $\llbracket [B] \rrbracket^1 \sigma = [c_1 \mapsto 0, c_2 \mapsto 0, c_3 \mapsto 2]$. By Definition 12, we have:

$$\begin{aligned} p_\sigma(c_1, c_3, 1) &= 4 & p_\sigma(c_2, c_3) &= 2 \\ p_\sigma(c_1, c_2, 1) &= 2 & p_\sigma(c_1, c_1) &= 1 & p_\sigma(c_2, c_2) &= 0. \end{aligned}$$

The following table represents the semantics at level 1 of our rules: rule₁ gives rise to transitions 1.-2., while rule₂ gives rise to transitions 3.-4.

| | | |
|-------------------|--------------------------------------|--------------------------------------|
| rule ₁ | 1. $\langle \{c_1, c_3\}, 4 \rangle$ | 2. $\langle \{c_2, c_3\}, 2 \rangle$ |
| rule ₂ | 3. $\langle \{c_1\}, 1 \rangle$ | 4. $\langle \{c_1, c_2\}, 2 \rangle$ |

It is worth noting that since $p_\sigma(c_2, c_2) = 0$, transition $\langle \{c_2\}, p_\sigma(c_2, c_2) \rangle$ does not belong to Tr_1 , and therefore cannot be in $\llbracket [A], [A], e \rrbracket_1^{\text{assoc-nc}}$. ■

The following lemma shows a relationship between the semantics at levels 0 and 1.

Lemma 5. Given an expression e and two patterns p_1 and p_2 , the following condition holds:

$$\llbracket p_1, p_2, e \rrbracket_1^{\text{assoc}} = \alpha_1^{\text{res}} \circ \llbracket p_1, p_2, e \rrbracket_0^{\text{assoc}} \circ \gamma_1^{\text{sys}}.$$

4.2 Abstraction at Level 2

In this section we propose an additional abstraction of Res_1 which calculates the cumulative propensity of all the reactions the assoc rule gives rise to. Abstraction of systems is the same one we introduced in the previous subsection.

Definition 14. A system $\sigma^\#$ is as a multiset of complexes, i.e., $\sigma^\# \in \text{Sys}_2 = \text{Sys}_1 = \text{mset Cplx}$. Moreover, we let $\text{Res}_2 = \mathbb{R}$.

We shall sometimes write σ instead of $\sigma^\#$ to stress the fact that $\text{Sys}_2 = \text{Sys}_1$. It is worth noting that both $\langle \wp(\text{Res}_1), \subseteq \rangle$ and $\langle \wp(\text{Res}_2), \subseteq \rangle$ are complete lattices. In the following we show that they are also related by a Galois connection [11].

Definition 15. We define the abstraction map $\alpha_2^{res} : \wp(\text{Res}_1) \rightarrow \wp(\text{Res}_2)$ and the concretization map $\gamma_2^{res} : \wp(\text{Res}_2) \rightarrow \wp(\text{Res}_1)$ as:

$$\alpha_2^{res}(\mathcal{R}) = \{ \sum_{\langle A,r \rangle \in \mathcal{R}} r \mid \mathcal{R} \in \mathbb{R} \} \quad \gamma_2^{res}(\mathcal{R}^\#) = \bigcup \{ \mathcal{R} \mid \alpha_2^{res}(\mathcal{R}) \subseteq \mathcal{R}^\# \}.$$

Lemma 6. $\langle \langle \wp(\text{Res}_1), \subseteq \rangle, \alpha_2^{res}, \gamma_2^{res}, \langle \wp(\text{Res}_2), \subseteq \rangle \rangle$ is a Galois connection.

We define $\llbracket p_1, p_2, e \rrbracket_2^{\text{assoc}}$, the semantics of the assoc rules at level 2.

Definition 16 (Level 2 semantics of assoc). Consider a state $\sigma \in \text{Sys}_2$ and a rule $\text{assoc } p_1 \ p_2 \ \text{rate } e \ \text{react } B$. We define the map $\llbracket p_1, p_2, e \rrbracket_2^{\text{assoc-nc}} : \text{Sys}_2 \rightarrow \text{Res}_2$ as

$$\llbracket p_1, p_2, e \rrbracket_2^{\text{assoc-nc}} \sigma = \sum_{\langle A,r \rangle \in \llbracket p_1, p_2, e \rrbracket_1^{\text{assoc-nc}} \sigma} r.$$

The collecting semantics $\llbracket p_1, p_2, e \rrbracket_2^{\text{assoc}} : \wp(\text{Sys}_2) \rightarrow \wp(\text{Res}_2)$ is defined as follows: $\llbracket p_1, p_2, e \rrbracket_2^{\text{assoc}} \Sigma = \{ \llbracket p_1, p_2, e \rrbracket_2^{\text{assoc-nc}} \sigma \mid \sigma \in \Sigma \}$.

The semantics of an assoc rule at level 1 determines all possible transitions Res_1 the rule gives rise to, and for each of them its propensity is calculated as well. At level 2 the rule's semantics determines only the cumulative propensity of the transitions obtained at level 1.

Example 10. Consider the rules introduced in Example 5 and the system σ defined in Example 6. Let us determine the semantics at level 2 of these rules in σ . In Example 9 we showed that the transitions obtained from rule₁ are $\langle \{c_1, c_3\}, 4 \rangle$ and $\langle \{c_2, c_3\}, 2 \rangle$, so their cumulative propensity is $4 + 2 = 6$. On the other hand, rule₂ gave transitions $\langle \{c_1\}, 1 \rangle$ and $\langle \{c_1, c_2\}, 2 \rangle$ and their cumulative propensity is $1 + 2 = 3$. Thus, the semantics at level 2 of rule₁ and rule₂ are 6 and 3 respectively.

Although this is a quite simple example, we can notice that there is an actual reduction of numbers of transitions appearing in different semantics of these two rules: rule₁ has 6 transitions Res_0 at level 0, 2 transitions Res_1 at level 1 and 1 transition Res_2 at level 2, while rule₂ has 3 transitions Res_0 at level 0, 2 transitions Res_1 at level 1 and 1 transition Res_2 at level 2. ■

In order to compute the semantics of an assoc rule at level 2, we should first determine the semantics of that rule at level 1 (Definition 16). Therefore, the complexity of the computation of the semantics at level 2 appears to be higher than the one of semantics at level 1. In some cases, however, we can compute the semantics at level 2 without computing the one at level 1, as shown in the following lemma. The proof relies on well-known combinatorial properties.

Lemma 7. *Let e be an expression with no occurrence of reactant_1 and reactant_2 , $\sigma \in \text{Sys}_2$ be a system. Then, the following equation holds:*

$$\llbracket p_1, p_2, e \rrbracket_2^{\text{assoc}} \sigma = \llbracket e \rrbracket \rho \cdot \left(\left(\llbracket p_1 \rrbracket^1 \sigma \right) \cdot \left(\llbracket p_2 \rrbracket^1 \sigma \right) - \left(\llbracket p_1 \rrbracket^1 \sigma \cap \llbracket p_2 \rrbracket^1 \sigma \right) - \left(\llbracket p_1 \rrbracket^1 \sigma \cap \llbracket p_2 \rrbracket^1 \sigma \right) \right).$$

Example 11. Let us consider one more time the system σ from Example 6 and the rule₂ from Example 5. We compute rule₂'s semantics at level 2 using Lemma 7. We showed in Example 8 that $\llbracket [A] \rrbracket^1 \sigma = [c_1 \mapsto 2, c_2 \mapsto 1, c_3 \mapsto 0]$, and therefore $\llbracket [A] \rrbracket^1 \sigma = 2 + 1 + 0 = 3$. We have $\llbracket [A], [A], 1 \rrbracket_2^{\text{assoc}} \sigma = 3 \cdot 3 - \binom{3}{2} - \binom{3}{1} = 3 \cdot 3 - \frac{3 \cdot 2}{2} - 3 = 3$, which is equal to the semantics of rule₂ computed by Definition 16 in Example 10. ■

5 Simulation

In this section we discuss how to simulate biological models expressed in our rule-based language. In doing that, we shall discuss how the abstractions provided in Sect. 4 can be exploited so to build optimized algorithms. In order to keep our presentation concise, we shall pretend that the model at hand is composed by assoc rules, only. Other kind of rules (dissoc, dyn) can indeed be handled through the same techniques.

5.1 Level 0 Simulation

We start by considering the problem of simulating a level 0 system, described via an initial state $\sigma^b \in \text{Sys}_0$ and a set of rules. A straightforward way to represent σ^b is that of storing the information relative to each complex $\langle n, c \rangle \in \sigma^b$ in its own memory area. That is, we allocate an ‘‘object’’ for every single complex in σ^b in which we store n and (a representation of) the multiset of the boxes in c , each one with its own state variables.

Simulating the system then can be done as follows. First, for each assoc rule, say indexed by $k \in K$, we compute $\llbracket [p_{k,1}], [p_{k,2}], e \rrbracket_0^{\text{assoc-nc}}$ by enumerating all the annotated complex pairs matching with the patterns. This provides us with sets of level 0 transitions $\{\langle A_{k,j}, a_{k,j} \rangle \mid j \in J\}$ (for some index set J), where each $A_{k,j}$ mentions exactly two annotated complexes. We assign to each transition the probability obtained by normalizing the rates (i.e., $a_{k,j} / \sum_{k,j} a_{k,j}$), and then randomly choose one of them, say $\langle A_{v,\mu}, a_{v,\mu} \rangle$. Simulation time is advanced by a random amount, generated according to the exponential distribution $\text{Exp}(\sum_{k,j} a_{k,j})$. The two annotated complexes in $A_{v,\mu}$ are removed from σ^b , then associated by merging their multisets of boxes (say c_1 and c_2). Finally we create a new annotated complex $\langle n', c_1 \cup c_2 \rangle$ for some fresh n' , and insert it in σ^b . At this point, the react code block of the rule is run (possibly modifying the newly created complex via its product variable, and spawning new complexes as well). The whole procedure is then repeated.

Below, we provide pseudo-code for the whole simulation procedure. We let index sets K, J to start counting from 1. Summing over the multi-index $\langle k, j \rangle \in K \times J$ follows the lexicographic ordering.

Level 0 simulation algorithm

- 1: $\{\text{assoc } [p_{k,1}][p_{k,2}] \text{ rate } e_k \text{ react } c_k \mid k \in K\} :=$ the set of rules of the model
- 2: $\sigma^b :=$ the initial state ; simulation time $t := 0$

- 3: **while** $t < t_{\max}$ **do**
- 4: **for all** rule indexes $k \in K$ **do**
- 5: compute the level 0 transitions $\{\langle A_{k,j}, a_{k,j} \rangle | j \in J\} := \llbracket [p_{k,1}], [p_{k,2}], e_k \rrbracket_0^{assoc-nc} \sigma^b$
- 6: **end for**
- 7: compute $a_0 := \sum_{k,j} a_{k,j}$; generate a random number $u := \mathcal{U}(0, a_0)$
- 8: find the minimum rule-transition index $\langle \nu, \mu \rangle$ such that $u \leq \sum_{\langle k,j \rangle = \langle 1,1 \rangle}^{\langle \nu, \mu \rangle} a_{k,j}$
- 9: $t := t + Exp(a_0)$; apply reaction μ by associating complexes in $A_{\nu,\mu}$, running command c_ν , and updating state σ^b accordingly
- 10: **end while**

The above simple algorithm is faithful to the continuous-time Markov chain which define the stochastic behaviour of biochemical systems. However, its space and time performance makes it unpractical for real-world applications. Indeed, one can note that it allocates a rather large amount of memory. This is because biochemical systems can involve a large number of complexes, hence allocating memory for each one of them should be avoided. Fortunately, in typical models it is often the case that many distinct complexes actually have identical state, so one can actually reduce the memory footprint by aggressively sharing the state data (and using copy-on-write to preserve the semantics). Even with this optimization, time performance suffers by the explicit enumeration of all possible reacting pairs. To overcome this problem, we abstract the system to level 1.

5.2 Level 1 Simulation

Lemma [1](#) states that, since an expression e can not access to the n component of a complex $\langle n, c \rangle$, evaluating e for distinct complexes sharing the same state yields the same value. Because of this, one can then evaluate it only once, and multiply the result by the number of complex pairs, hence obtaining the cumulative rate for all such transitions. In order to do that, we do not need to actually enumerate all the complex pairs, but just to compute their number, which can easily be done exploiting combinatorial formulae while keeping track of the amount of complexes in each state, i.e. counting the population for each species. This greatly improves the time performance of the simulation algorithm.

Since we now need only a population count, we can avoid to store the names n for each complex, so to further improve the memory footprint. This essentially amount to move to the level 1 abstraction, i.e. turning $\sigma^b \in Sys_0$ into a $\sigma \in Sys_1$. Simulating at that level results in a less detailed simulation output, which describes which species interact without specifying the actual identities of the involved complexes. Since identities are unimportant from a biological point of view, and quantities are, the result of simulation still preserves all the relevant information of level 0.

Below, we adapt the level 0 simulation algorithm so to work at level 1. This actually results in the well-known Gillespie's Direct Method [\[4\]](#), which indeed works precisely at that abstraction level. A minor difference worth mentioning is that in our models an unbounded number of new chemical species (i.e., complexes) can be formed during simulation, while in the reaction-based models considered by Gillespie the set of such species is finite and statically known before simulation is started.

Level 1 simulation algorithm (Gillespie's Direct Method)

- 1: {assoc $[p_{k,1}][p_{k,2}]$ rate e_k react c_k mid $k \in K$ } := the set of rules of the model
- 2: σ := the initial state ; simulation time $t := 0$
- 3: **while** $t < t_{\max}$ **do**
- 4: **for all** rule indexes $k \in K$ **do**
- 5: compute the level 1 transitions $\langle A_{k,j}, a_{k,j} \rangle | j \in J := \llbracket [p_{k,1}], [p_{k,2}], e_k \rrbracket_1^{assoc-nc} \sigma$
exploiting Def. [12](#) to compute propensities $a_{k,j}$
- 6: **end for**
- 7: compute $a_0 := \sum_{k,j} a_{k,j}$; generate a random number $u := \mathcal{U}(0, a_0)$
- 8: find the minimum rule-transition index $\langle \nu, \mu \rangle$ such that $u \leq \sum_{\langle k,j \rangle = \langle 1,1 \rangle}^{(\nu, \mu)} a_{k,j}$
- 9: $t := t + Exp(a_0)$; apply reaction μ by associating complexes in $A_{\nu, \mu}$, running command c_ν , and updating state σ accordingly
- 10: **end while**

Comparing the above to the level 0 algorithm, we find the main, important difference in line [5](#), where we exploit the combinatorial formula in Def. [12](#) to compute propensities.

Several further standard optimizations can be applied, e.g., after a transition has been applied, we can avoid to recompute those propensities which are known to be unaffected by that transition. This is typically done via a dependency graph [3](#).

5.3 Level 2 Simulation

We saw how abstracting the model from level 0 to level 1 improves the space and time performance of the simulation. We now investigate the consequences of further abstracting the model to level 2. As we shall see, under some assumptions, this may lead to further improvements in time performance.

Recall that level 2 systems $\sigma^\# \in Sys_2$ are actually identical to level 1 systems $\sigma \in Sys_1$, so no information about the complexes is actually lost here. Instead, level 2 abstract the transitions which are being generated by the semantics. More in detail, we have that *all* the level 1 transitions ($\in Res_1$) which are being generated by any given rule are collapsed into a single level 2 transition ($\in Res_2$) having as its rate the sum of all the rates of level 1 transitions. In this way, the propensities of level 1 transitions are combined to form a single *cumulative propensity* for the rule at hand. Also, in the common case in which the rate expression of such rule is simply a constant, the cumulative propensity for the rule can be computed efficiently exploiting Lemma [7](#). This suggests a possible modification to the level 1 simulation algorithm (Direct Method) which we provide below.

Level 2 simulation algorithm

- 1: {assoc $[p_{k,1}][p_{k,2}]$ rate e_k react c_k | $k \in K$ } := the set of rules of the model
- 2: $\sigma^\#$:= the initial state ; simulation time $t := 0$
- 3: **while** $t < t_{\max}$ **do**
- 4: **for all** rule indexes $k \in K$ **do**
- 5: compute the level 2 transition $r_k := \llbracket [p_{k,1}], [p_{k,2}], e_k \rrbracket_2^{assoc-nc} \sigma^\#$ exploiting the formula in Lemma [7](#)
- 6: **end for**

- 7: compute $r_0 := \sum_{k \in K} r_k$ and generate a random number $u := \mathcal{U}(0, r_0)$
- 8: find the minimum rule-index ν such that $u \leq \sum_{k=1}^{\nu} r_k$; let $u := u - \sum_{k=1}^{\nu-1} r_k$
- 9: compute the level 1 transitions $\{\langle A_j, a_j \rangle \mid j \in J\} := \llbracket [p_{\nu,1}], [p_{\nu,1}], e_{\nu} \rrbracket_1^{assoc-nc} \sigma^{\#}$ exploiting Def. 12 to compute propensities a_j
- 10: find the minimum transition index μ such that $u \leq \sum_{j=1}^{\mu} a_j$
- 11: $t := t + Exp(r_0)$; apply reaction μ by associating complexes in A_{μ} , running command c_{ν} and updating state $\sigma^{\#}$ accordingly
- 12: **end while**

The main change with respect to level 1 can be summarized as follows. In level 1 simulation, we generate all the level 1 transitions and then randomly pick among them. In level 2 simulation, we only generate *one level 2 transition per rule* from which we randomly pick one. After such choice is done, we know the rule ν which is to be applied: we then generate level 1 transitions for that rule *only*, and then pick among those. The net result is that we perform two random choices among two small sets instead of one choice in a large set. Assume for the sake of illustration that a model features 20 association rules, and that each pattern in them matches with 5 complexes. Generating all the level 1 transitions requires enumerating all the $\approx 20 \cdot 5 \cdot 5 = 500$ cases. Instead, performing two separate choices for level 2 and level 1 transition requires enumerating only $\approx 20 + 5 \cdot 5 = 45$ cases.

In order to exploit the observation above, it is important to exploit Lemma 7 to compute level 2 transitions efficiently. For that, we need to quickly compute, for each association rule, the quantities $\llbracket [p_1] \rrbracket^1 \sigma$, $\llbracket [p_2] \rrbracket^1 \sigma$, and $\llbracket [p_1] \rrbracket^1 \sigma \cap \llbracket [p_2] \rrbracket^1 \sigma$. This can be done by keeping track for each rule of three sets of complexes: those matching with p_1 , p_2 , and both. These sets need to be updated infrequently: updates are needed only when a complex c is added to a system σ for which $\sigma(c) = 0$, i.e., when the first copy of c appears in the system. Having these three sets, computing the wanted cardinalities is done by summing all the populations of the complexes. An incremental approach which adjusts such quantities at every step – without recomputing them – is also feasible.

Steps number 9 and 10 can be further optimized. There, we find the set of complexes A_{μ} to be associated by generating all the level 1 transitions for rule ν , and then using their rates as weights for the random choice of A_{μ} . This might be improved by using an alternative way to extract A_{μ} from the same distribution, so to avoid the expensive enumeration of all the level 1 transitions. One such way is as follows. Randomly extract a complex c_1 from those matching with pattern $p_{\nu,1}$, using their population count as weights. Complex c_2 can be chosen similarly using $p_{\nu,2}$. Here, however, in the case c_1 also matches with $p_{\nu,2}$, we decrement the population of c_1 by one unit. This adjustment reflects the fact that association must involve two *distinct* annotated complexes. This alternative way of extracting $A_{\mu} = \{c_1, c_2\}$ is advantageous since it requires at most a linear scan of all the complexes matching p_1 and p_2 . By contrast, enumerating the level 1 transitions can generate a quadratic number of them.

6 Conclusions

We introduced \mathcal{L} , a rule-based imperative language for the modeling and simulation of biochemical systems. We provided a concrete semantics for it, as well as two abstract

ones. We then exploited the abstractions so to devise efficient simulation algorithms for \mathcal{L} . Future work will investigate extensions of \mathcal{L} to more specific kinds of models, e.g. those involving compartments or other formalisms to represent space.

References

1. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proceedings of the 4th POPL, pp. 238–252. ACM (1977)
2. Danos, V., Feret, J., Fontana, W., Harmer, R., Krivine, J.: Rule-Based Modelling of Cellular Signalling. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 17–41. Springer, Heidelberg (2007)
3. Gibson, M.A., Bruck, J.: Efficient exact stochastic simulation of chemical systems with many species and many channels. *Journal of Physical Chemistry A* 104(9), 1876–1889 (2000)
4. Gillespie, D.T.: Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry* 81(25), 2340–2361 (1977)
5. Priami, C., Quaglia, P., Romanel, A.: BlenX Static and Dynamic Semantics. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 37–52. Springer, Heidelberg (2009)
6. Priami, C., Quaglia, P., Zunino, R.: An imperative language of self-modifying graphs for biological systems. In: ACM Symposium on Applied Computing (SAC) (to appear, 2012)
7. Romanel, A., Priami, C.: On the decidability and complexity of the structural congruence for beta-binders. *Theor. Comput. Sci.* 404(1-2), 156–169 (2008)

Sound Control-Flow Graph Extraction for Java Programs with Exceptions

Afshin Amighi¹, Pedro de C. Gomes², Dilian Gurov², and Marieke Huisman¹

¹ University of Twente, Enschede, The Netherlands

² KTH Royal Institute of Technology, Stockholm, Sweden

Abstract. We present an algorithm to extract control-flow graphs from Java bytecode, considering exceptional flows. We then establish its correctness: the behavior of the extracted graphs is shown to be a sound over-approximation of the behavior of the original programs. Thus, any temporal safety property that holds for the extracted control-flow graph also holds for the original program. This makes the extracted graphs suitable for performing various static analyses, in particular model checking. The extraction proceeds in two phases. First, we translate Java bytecode into BIR, a stack-less intermediate representation. The BIR transformation is developed as a module of Sawja, a novel static analysis framework for Java bytecode. Besides Sawja's efficiency, the resulting intermediate representation is more compact than the original bytecode and provides an explicit representation of exceptions. These features make BIR a natural starting point for sound control-flow graph extraction. Next, we formally define the transformation from BIR to control-flow graphs, which (among other features) considers the propagation of uncaught exceptions within method calls. We prove the correctness of the two-phase extraction by suitably combining the properties of the two transformations with those of an idealized control-flow graph extraction algorithm, whose correctness has been proved directly. The control-flow graph extraction algorithm is implemented in the CONFLEX tool. A number of test-cases show the efficiency and the utility of the implementation.

1 Introduction

Over the last decade, there has been a steadily increasing demand for software quality and reliability. Different formal techniques have been deployed to reach this goal, such as various static analyses, model checking and (automated) theorem proving. A major obstacle for the application of formal techniques is that the state space of software is typically infinite. Appropriate abstractions are thus necessary in order to make the formal analyses tractable. Further, it is important that such abstractions are *sound* w.r.t. the original program: if a property holds over the abstract model, it should also be a property of the original program.

A common approach is to generate an abstract model from the code, only preserving the information that is relevant for the class of properties of interest. In particular, *control-flow graphs* (CFGs) are a widely used abstraction, where

only the control-flow information is kept, and all program data is abstracted away (see *e.g.* [6,19,16]). In a CFG, nodes represent the control points of the program, while edges represent the instructions that move control between control points.

Numerous techniques have been proposed to extract automatically control-flow graphs from program code (see *e.g.* [15,8,16]). Typically, however, these are not accompanied by a formal soundness argument. The present paper attempts to fill this gap: we define a control-flow graph extraction algorithm for sequential Java bytecode (JBC), and show that the extraction algorithm is sound w.r.t. the behavior (*i.e.*, executions) of the program. The extraction algorithm considers all the typical intricacies of Java, such as virtual method call resolution, the differences between dynamic and static object types, and exception handling. In particular, it includes explicitly thrown instructions, and a significant subset of run-time exceptions. The sound analysis of exceptional flows is particularly challenging for two reasons. First, the stack-based nature of the Java Virtual Machine (JVM) makes it hard to statically determine the type of explicitly thrown exceptions, thus making it difficult to decide to which handler (if any) control will be transferred. Second, the JVM can raise (implicit) run-time exceptions, such as *NullPointerException* and *IndexOutOfBoundsException*, and to keep track of where such exceptions can be raised requires much care.

We present a two-phase extraction algorithm using the Bytecode Intermediate Representation (BIR) language [9], developed by Demange *et al.* The use of BIR has several advantages. First of all, BIR provides a stack-less representation of JBC. Thus, all instructions (including the explicit `throw`) are directly connected with their operands. This allows to determine the static type of explicitly thrown exceptions. In addition, the representation of a program in BIR is smaller, since operations are not stack-based, but represented as expression trees. Second, BIR supports the analysis of implicitly thrown exceptions by generating assertions that indicate when the next instruction might raise a run-time exception, following the approach proposed for the Jalapeño compiler [7]. Finally, Demange *et al.* present formal translation rules from JBC, and define an operational semantics for BIR. They show that the resulting program is semantics-preserving with respect to observable events, such as raising exceptions, and sequences of method invocations. This result increases the reliability of the correctness of the BIR transformation, and in consequence, also of our CFG extraction algorithm.

Our two-phase extraction algorithm first uses the transformation from JBC to BIR from Sawja [11], a library for static analysis of Java bytecode, and then it extracts CFGs from BIR. It is implemented as the tool CONFLEX. Sawja provides only intra-procedural support for exceptions. Thus, to obtain a sound extraction tool, on top of this we implemented a fixed-point computation of exceptional flow caused by uncaught exceptions.

Proving correctness of the two-phase extraction algorithm directly (*e.g.*, by means of behavioral simulation) is cumbersome. Instead, we use the correctness of an idealized direct extraction algorithm by Amighi [2,3] to simplify the overall correctness argument. We connect the CFGs that are extracted by the idealized algorithm and by the two-phase algorithm via a (structural) simulation relation,

and use a previous result (see [10, Th. 36]) to infer behavioral simulation. From this, one can conclude that all behaviors of the CFG generated by the indirect algorithm (BIR) are a sound over-approximation of the original program behavior. Thus, the extraction algorithm produces control-flow graphs that are sound for the verification of temporal safety properties. We outline the correctness proof in Section 4; the details can be found in an accompanying technical report [3].

Organization. The remainder of this paper is organized as follows. Section 2 provides the necessary definitions for the algorithm and its correctness proof. Section 3 presents the two-phase extraction algorithm, its implementation, and experimental evaluation. In Section 4 we discuss the correctness of the algorithm. Finally, in Section 5 we discuss related work, and conclude with Section 6.

2 Preliminaries

Control-flow graphs (CFGs) provide an abstract model of programs. Method graphs are the basic building blocks of CFGs. Let METH and EXCP be two countably infinite sets of method names and exception names, respectively. Method graphs are defined as Kripke structures, as follows.

Definition 1 (Method Graph). *A method graph for method m over given finite sets $M \subseteq \text{METH}$ and $E \subseteq \text{EXCP}$ is a pair $(\mathcal{M}_m, \mathbb{E}_m)$, where $\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$ is a transition-labeled Kripke structure, and $\mathbb{E}_m \subseteq V_M$ is a non-empty set of entry points of m . V_m is the set of control points of m , $L_m = M \cup \{\varepsilon, \text{handle}\}$ is the set of transition labels, $\rightarrow_m \subseteq V_m \times L_m \times V_m$ is the labeled transition relation between control points, $A_m = \{m, r\} \cup E$ is the set of atomic propositions, and $\lambda_m : V_m \rightarrow \mathcal{P}(A_m)$ is a valuation function such that $m \in \lambda_m(v)$ for all $v \in V_m$, and for all $x, x' \in E$, if $x, x' \in \lambda_m(v)$ then $x = x'$, i.e., each control node is valued with the method signature it belongs to, and with at most one exception.*

A node $v \in V_m$ is marked with the atomic proposition r whenever it is a return node of the method. Internal transfer edges are labeled with ε , and the control transfers caused by the handling of exceptions are labeled with *handle*. All other edges correspond to method calls, and are labeled with the called method.

The *control-flow graph* of a program is simply the disjoint union of all method graphs of methods defined in the program. Figure 1 shows an example Java program with two methods, and a corresponding CFG. Every control-flow graph is equipped with an *interface* $I = (I^+, I^-, E')$, defining the methods that are provided to and required from the environment, denoted by $I^+, I^- \subseteq M$, and the exceptions that may be raised by each method, but not caught, indicated as $E' \subseteq E$. If $I^- \subseteq I^+$ then I is *closed*.

We use a standard notion of control-flow graph *behavior* based on pushdown automata, where configurations are pairs of control nodes and stacks of method invocations. Internal transitions are labeled with τ for normal transfers, *throw x* and *catch x* for exceptional transfers, m_1 *call* m_2 and m_1 *ret* m_2 for normal

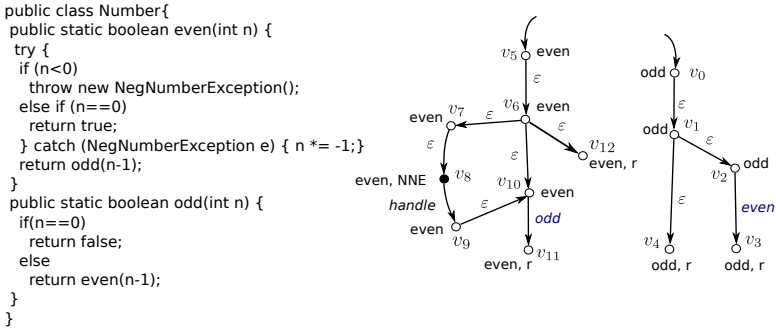


Fig. 1. An example program and its control-flow graph

inter-procedural transfers, and m_1 *xret* m_2 for returns caused by an uncaught exception. The formal definition is straightforward (see [12]), and its details are not necessary to understand the correctness proof of our extraction algorithm.

3 Extracting Control-Flow Graphs from BIR

This section presents the two-phase transformation from Java bytecode into control-flow graphs using BIR as intermediate representation. First, we briefly present the BIR language, and its transformation function from JBC, named BC2BIR. Next, we present how BIR is transformed into CFGs. We conclude by describing the implementation of the algorithm as the CONFLEX tool [1], and presenting some experimental results.

3.1 The BIR Language

The BIR language is an intermediate representation of Java bytecode. The main difference with JBC is that BIR instructions are stack-less, in contrast to bytecode instructions that operate over values stored on the operand stack. We give a brief overview of BIR; for a full account we refer to [9].

Figure 2 summarizes the BIR syntax. Its instructions operate over expression trees, i.e., arithmetic expressions composed of constants, operations, variables, and fields of other expressions ($expr.f$). BIR does not have operations over strings and booleans; these are transformed into method calls by the BC2BIR transformation. It also reconstructs expression trees, i.e., it collapses one-to-many stack-based operations into a single expression. As a result, a program represented in BIR typically has fewer instructions than the original JBC program.

BIR has two types of variables: *var* and *tvar*. The first are identifiers also present in the original bytecode; the latter are new variables introduced by the transformation. Both variables and object fields can be an assignment’s target.

Many of the BIR instructions have an equivalent JBC counterpart, e.g., `nop`, `goto` and `if`. A `return expr` ends the execution of a method with return value

| | |
|--|---|
| $ \begin{aligned} \text{expr} ::= & c \mid \text{null} && (\text{constants}) \\ & \mid \text{expr} \oplus \text{expr} && (\text{arithmetic}) \\ & \mid \text{tvar} \mid \text{lvar} && (\text{variables}) \\ & \mid \text{expr.f} && (\text{field access}) \\ \text{lvar} ::= & l \mid l_1 \mid l_2 \mid \dots && (\text{local var.}) \\ & \text{this} \\ \text{tvar} ::= & t \mid t_1 \mid t_2 \mid \dots && (\text{temp. var.}) \\ \text{target} ::= & \text{lvar} \\ & \mid \text{tvar} \\ & \mid \text{expr.f} \end{aligned} $ | $ \begin{aligned} \text{Assignment} ::= & \text{target} := \text{expr} \\ \text{Return} ::= & \text{return expr} \mid \text{return} \\ \text{MethodCall} ::= & \text{expr.ns}(\text{expr}, \dots, \text{expr}) \\ & \mid \text{target} := \text{expr.ns}(\text{expr}, \dots, \text{expr}) \\ \text{NewObject} ::= & \text{target} := \text{new } C(\text{expr}, \dots, \text{expr}) \\ \text{Assertion} ::= & \text{nonnull expr} \mid \text{notzero expr} \\ \text{instr} ::= & \text{nop} \mid \text{if expr pc} \mid \text{goto pc} \\ & \mid \text{throw expr} \mid \text{mayinit } C \\ & \mid \text{Assignment} \mid \text{Return} \\ & \mid \text{MethodCall} \mid \text{NewObject} \\ & \mid \text{Assertion} \end{aligned} $ |
|--|---|

Fig. 2. Expressions and Instructions of BIR

| <i>Assertion</i> | <i>Exception</i> | <i>Assertion</i> | <i>Exception</i> |
|------------------|-----------------------------------|------------------|----------------------------|
| [nonnull] | <i>NullPointerException</i> | [notzero] | <i>ArithmeticException</i> |
| [checkbound] | <i>IndexOutOfBoundsException</i> | [checkcast] | <i>ClassCastException</i> |
| [notneg] | <i>NegativeArraySizeException</i> | [checkstore] | <i>ArrayStoreException</i> |

Fig. 3. Implicit exceptions supported by BIR, and associated assertions

expr, while **return** ends a *void* method. The **throw** instruction explicitly transfers control flow to the exception handling mechanism. Method call instructions are represented by their method signature. For non-*void* methods, the instruction assigns the result value to a variable.

In contrast to JBC, object allocation and initialization are done in a single step, during execution of the **new** instruction. Java also has class initialization, i.e., the one-time initialization of a class's static fields. BIR has the special instruction **mayinit** to indicate that at that point a class may be initialized for the first time. Otherwise, it behaves exactly as **nop**.

BIR's support of implicit exceptions follows the approach proposed for the Jalapeño compiler [7]. It inserts special assertions before the instructions that can potentially raise an exception, as defined by the JVM. Figure 3 shows all implicit exceptions that are currently supported by the BC2BIR transformation [5], and the associated assertion. For example, the transformation inserts a **[nonnull]** assertion before any instruction that might raise a *NullPointerException*, such as an access to a reference. If the assertion holds, it behaves as a **[nop]**, and control-flow passes to the next instruction. If the assertion fails, control-flow is passed to the exception handling mechanism. In the transformation from BIR to CFG, we use a function χ to obtain the exception associated with an instruction (as presented in Figure 3). Notice that our translation from BIR to CFG can easily be adapted to other implicit exceptions, provided appropriate assertions are generated for them.

A BIR program is organized in exactly the same way as a Java bytecode program. A program is a set of classes, ordered by a class hierarchy. Every class consists of a name, methods and fields. Methods have code, stored in an

| Input | Output | Input | Output | Input | Output |
|------------------|--|---------|------------|------------|------------------|
| pop | \emptyset | nop | [nop] | div | [notzero e_2] |
| push c | \emptyset | if p | [if e pc'] | athrow | [throw e] |
| dup | \emptyset | goto p | [goto pc'] | new C | [mayinit C] |
| load x | \emptyset | return | [return] | getfield f | [nonnull e] |
| add | \emptyset | vreturn | [return e] | | |
| Input | Output | | | | |
| store x | [x:=e] or [$t_{pc}^0 := x; x := e$] | | | | |
| putfield f | [nonnull e; $FSave(pc, f, as); e.f := e'$] | | | | |
| invokevirtual ns | [nonnull e; $HSave(pc, as); t_{pc}^0 := e.ns(e'_1 \dots e'_n)$] | | | | |
| invokespecial ns | [nonnull e; $HSave(pc, as); t_{pc}^0 := e.ns(e'_1 \dots e'_n)$] or [$HSave(pc, as); t_{pc}^0 := new C(e'_1 \dots e'_n)$] | | | | |

Fig. 4. Rules for $BC2BIR_{instr}$

instruction array, with indexing starting with 0 for the entry control point. However, in contrast to JBC, in BIR the indexes in the instruction array are sequential.

3.2 Transformation from Java Bytecode into BIR

Next we briefly describe the $BC2BIR$ transformation. It translates a complete JBC program into BIR by symbolically executing the bytecode using an abstract stack. This stack is used to reconstruct expression trees, and to connect instructions to its operands. As we are only interested in the set of BIR instructions that can be produced, we do not discuss all details of this transformation. For the complete algorithm, we refer to [9].

The symbolic execution of the individual instructions is defined by a function $BC2BIR_{instr}$ that, given a program counter, a JBC instruction and an abstract stack, outputs a sequence of BIR instructions and a modified abstract stack. In case there is no match for a pair of bytecode instruction and stack, the function returns the *Fail* element, and the $BC2BIR$ algorithm aborts.

Definition 2 (BIR Transformation Function). *Let $AbsStack \in expr^*$. The rules defining the instruction-wise transformation $BC2BIR_{instr} : \mathbb{N} \times instr_{JBC} \times AbsStack \rightarrow ((instr_{BIR})^* \times AbsStack) \cup \{Fail\}$ from Java bytecode into BIR are given in Figure 4.*

As a convention, we use brackets to distinguish BIR instructions from their JBC counterparts. Variables t_{pc}^i are new, introduced by the transformation.

JBC instructions `if`, `goto`, `return` and `vreturn` are transformed into corresponding BIR instructions. The `new` instruction is distinct from `[new C()]` in BIR, and produces a `[mayinit]`. The `getfield f` instruction reads a field from the object reference at the top of the stack. This might raise a *NullPointerException*, therefore the transformation inserts a `[nonnull]` assertion.

Instruction `store x` produces one or two assignments, depending on the state of the abstract stack. Instruction `putfield f` outputs a set of BIR instructions: `[nonnull e]` guards if e is a valid reference; the auxiliary function $FSave$

| | |
|--|---|
| 0: <code>iload_0</code> | |
| 1: <u><code>ifne 6</code></u> | 0: <code>if (n != 0) goto 2</code> |
| 4: <code>iconst_0</code> | |
| 5: <u><code>ireturn</code></u> | 1: <code>return 0</code> |
| 6: <code>aload_0</code> | |
| 7: <code>iconst_1</code> | |
| 8: <code>isub</code> | 2: <code>mayinit Number</code> |
| 9: <u><code>invokestatic Number.even(int)</code></u> | 3: <code>t₃⁰ := Number.even(n - 1)</code> |
| 12: <u><code>ireturn</code></u> | 4: <code>return t₃⁰</code> |

Fig. 5. Comparison between instructions in method `odd()` in JBC and BIR

generates a sequence of assignments to temporary variables; followed by the assignment to the field `e.f`. Similarly, `invokevirtual` generates a `[nonnull]` assertion, followed by a set of assignments to temporary variables – represented as the auxiliary function `HSave` – and the call instruction itself. The transformation of `invokespecial` can produce two different sequences of BIR instructions. The first case is the same as for `invokevirtual`. In the second case, there are assignments to temporary variables (`HSave`), followed by the instruction `[new C]`, which denotes a call to the constructor.

Figure 5 shows the JBC and BIR versions of method `odd()` from Figure 1. The different colors show the collapsing of JBC instructions by the transformation; the underlined instructions are the ones that produce BIR instructions. The BIR method has a local variable (`n`) and a newly introduced variable (`t30`). Notice that the argument for the method invocation and the operand to the `[if]` instruction are reconstructed expression trees. The `[mayinit]` instruction shows that class `Number` may be initialized in that program point.

3.3 Transformation from BIR into Control-Flow Graphs

The extraction algorithm that generates a CFG from BIR iterates over the instructions of a method. It uses the transformation function $\mathbf{b}\mathcal{G}$, that takes as input a program counter and instruction from a BIR method, plus its exception table. Each iteration outputs a set of edges.

To define $\mathbf{b}\mathcal{G}$, we introduce auxiliary definitions. First, let \mathcal{E}_t be the set of all exception tables. $H \in \mathcal{E}_t$ is the exception table for the given method, containing the same entries as the JBC table, but with control points relating to BIR instructions. The function $h_H(\mathbf{pc}, x)$ searches for the first handler for the exception x (or a subtype) at position \mathbf{pc} . The function $\mathcal{H}_x^{\mathbf{pc}}$ returns one edge after querying h_H : if there was an exception handler, it returns an edge to a normal control node; otherwise, it returns an edge to an exceptional return node.

The extraction is parametrized by a virtual method call resolution algorithm α . The function $res^\alpha(\mathbf{ns})$ uses α to return a safe over-approximation of the possible receivers to a virtual method call with signature \mathbf{ns} , or the single receiver if the signature is from a non-virtual method (e.g. a static method).

$$\begin{aligned}
\mathcal{H}_x^{\text{pc}} &= \begin{cases} \{ (\bullet_m^{\text{pc},x}, \text{handle}, \circ_m^{\text{pc}'}) \} & \text{if } h_H(\text{pc}, x) = \text{pc}' \neq 0 \\ \{ (\bullet_m^{\text{pc},x}, \text{handle}, \bullet_m^{\text{pc},x,r}) \} & \text{if } h_H(\text{pc}, x) = 0 \end{cases} \\
\text{bG}(i_{\text{pc}}, H) &= \begin{cases} \{ (\circ_m^{\text{pc}}, \varepsilon, \circ_m^{\text{pc}+1}) \} & \text{if } i \in \text{Assignment} \cup \{ [\text{nop}], [\text{mayinit}] \} \\ \{ (\circ_m^{\text{pc}}, \varepsilon, \circ_m^{\text{pc}+1}), (\circ_m^{\text{pc}}, \varepsilon, \circ_m^{\text{pc}'}) \} & \text{if } i = [\text{if expr pc}'] \\ \{ (\circ_m^{\text{pc}}, \varepsilon, \circ_m^{\text{pc}'}) \} & \text{if } i = [\text{goto pc}'] \\ \{ (\circ_m^{\text{pc}}, \varepsilon, \circ_m^{\text{pc},r}) \} & \text{if } i \in \text{Return} \\ \bigcup_{x \in X} \{ (\circ_m^{\text{pc}}, \varepsilon, \bullet_m^{\text{pc},x}) \} \cup \mathcal{H}_x^{\text{pc}} & \text{if } i = [\text{throw } X] \\ \{ (\circ_m^{\text{pc}}, \varepsilon, \circ_m^{\text{pc}+1}), (\circ_m^{\text{pc}}, \varepsilon, \bullet_m^{\text{pc},\chi(i)}) \} \cup \mathcal{H}_{\chi(i)}^{\text{pc}} & \text{if } i \in \text{Assertion} \\ \{ (\circ_m^{\text{pc}}, \mathbf{C}, \circ_m^{\text{pc}+1}), (\circ_m^{\text{pc}}, \varepsilon, \bullet_m^{\text{pc},eN}) \} \cup \mathcal{H}_{eN}^{\text{pc}} \cup \mathcal{N}_{\mathbf{C}}^{\text{pc}} & \text{if } i \in \text{NewObject} \\ \{ (\circ_m^{\text{pc}}, n, \circ_m^{\text{pc}+1}) \} \cup \mathcal{N}_n^{\text{pc}} & \text{if } i \in \text{MethodCall} \end{cases} \\
\mathcal{N}_n^{\text{pc}} &= \bigcup_{\bullet_n^{\text{pc}',x,r} \in \text{bG}(n)} \{ (\circ_m^{\text{pc}}, \text{handle}, \bullet_m^{\text{pc},x}) \} \cup \mathcal{H}_x^{\text{pc}}
\end{aligned}$$

Fig. 6. Extraction rules for control-flow graphs from BIR

We divide the definition of bG into two parts. The *intra-procedural* analysis extracts for every method an initial CFG, based solely on its instruction array, and its exception table. Based on these CFGs, the *inter-procedural* analysis computes the functions $\mathcal{N}_n^{\text{pc}}$, which return exceptional edges for exceptions propagated by calls to method n . The functions for inter-dependent methods are thus mutually recursive, and are computed in a fixed-point manner.

Definition 3 (Control Flow Graph Extraction). *The control-flow graph extraction function $\text{bG} : (\text{Instr} \times \mathbb{N}) \times \mathcal{E}_t \rightarrow \mathcal{P}(V \times L_m \times V)$ is defined by the rules in Figure 6. Given method m , with ArInstr_m as its instruction array, the control-flow graph for m is defined as $\text{bG}(m) = \bigcup_{i_{\text{pc}} \in \text{ArInstr}_m} \text{bG}(i_{\text{pc}}, H_m)$, where i_{pc} denotes the instruction with array index pc . Given a closed BIR program Γ_B , its control-flow graph is $\text{bG}(\Gamma_B) = \bigcup_{m \in \Gamma_B} \text{bG}(m)$.*

First, we describe the rules for the intra-procedural analysis. Assignments, `[nop]` and `[mayinit]` add a single edge to the next normal control node. The conditional jump `[if expr pc']` produces a branch in the CFG: control can go either to the next control point, or to the branch point pc' . The unconditional jump `goto pc'` adds a single edge to control point pc' . The `[return]` and `[return expr]` instructions generate an internal edge to a return node, i.e., a node with the atomic proposition r . Notice that, although both nodes are tagged with the same pc , they are different, because their sets of atomic propositions are different.

The `[throw X]` instruction, similarly to virtual method call resolution, depends on a static analysis to find out the possible exceptions that can be thrown. The BIR transformation only provides the static type X of the thrown exception. Let X also denote the set containing the static type, and all its subtypes. The transformation produces an exceptional edge for each element x of X , followed by the appropriate edge derived from the exception table.

The rule for assertion instructions produces a normal edge, for the case that the implicit exception is not raised, and an edge to the exceptional node tagged with the exception type (as defined in Figure 3), together with the appropriate edge derived from the exception table.

The extraction rule for a constructor call (`[new C]`) produces a single normal edge, since there is only one possible receiver for the call. In addition, we also produce an exceptional edge, because of a possible *NullPointerException*. The rule for the other method invocations adds a single normal edge for each possible receiver n returned by res^α .

Next, we describe the inter-procedural analysis. In all program points where there is a method invocation, the function $\mathcal{N}_n^{\text{pc}}$ adds exceptional edges, relative to the exceptions that are propagated by method calls. It checks if the CFG of an invoked method n contains an exceptional return node. If it does, then function $\mathcal{H}_x^{\text{pc}}$ verifies whether the exception is caught upon return. If so, it adds an edge to the handler. Otherwise it adds an edge to an exceptional return node. In the latter case, propagation of the exception continues until it is caught by a caller method, or there are no more methods to handle it. This is similar to the process described by Jo and Chang [16], who also present a fixed-point algorithm to compute the propagation edges. It checks the pre-computed call-graph which are the callers to a method propagating a given exception, and at which control-points. If there is a suitable handler for that exception, it adds the respective handling edges, and the process stop. Otherwise, the computation proceeds.

3.4 Implementation

The extraction rules from Figure 6 are implemented in our CFG extraction tool CONFLEX. It uses Sawja for the transformation from bytecode into BIR, and for virtual method call resolution. Sawja supports several resolution algorithms. Experimental evaluation showed that the algorithm’s choice impacts the performance, but does not affect significantly the precision. Table 1 shows the results using Rapid Type Analysis [4], which presented the best balance between time and precision [21]. The table provides statistics for the CFG extraction of several examples with varying sizes. All experiments are done on a server with an Intel i5 2.53 GHz processor and 4GB of RAM. Methods from the API are not extracted; only classes that are part of the program are considered.

BIR Time is the time spent to transform JBC into BIR. For the transformation from BIR to CFG, we provide statistics for the intra-procedural and the inter-procedural analysis.

Table 1 shows that in all cases the number of BIR instructions is less than 40% of the JBC instructions. This indicates that the use of BIR mitigates the blow-up of control-flow graphs, and clearly program analysis benefits from this. The computation time for intra- and inter-procedural analysis grows proportionally with the number of BIR instructions. The intra-procedural analysis is linear w.r.t. to the number of instructions, and the experimental results of the inter-procedural analysis show that it only contributes to a small part of the total extraction time.

Table 1. Statistics for CONFLEx

| Software | # of JBC instr. | # of BIR instr. | BIR time (ms) | Intra-Procedural | | | Inter-Procedural | | |
|-------------|-----------------|-----------------|---------------|------------------|------------|-----------|------------------|------------|-----------|
| | | | | # of nodes | # of edges | time (ms) | # of nodes | # of edges | time (ms) |
| Jasmin | 30930 | 10850 | 267 | 19152 | 19460 | 320 | 21651 | 21966 | 25 |
| JFlex | 53426 | 20414 | 706 | 38240 | 38826 | 859 | 42442 | 43072 | 23 |
| Groove Ima. | 193937 | 77620 | 587 | 159046 | 158593 | 4817 | 193268 | 192905 | 1849 |
| Groove Gen. | 328001 | 128730 | 926 | 251762 | 252102 | 13609 | 308164 | 308638 | 5541 |
| Groove Sim. | 427845 | 167882 | 1072 | 311008 | 311836 | 16067 | 386553 | 387556 | 6886 |
| Soot | 1345574 | 516404 | 98692 | 977946 | 976212 | 264690 | 1209823 | 1208358 | 57621 |

We do not provide comparative data with other extraction tools, such as Soot [22], or Wala [14] because this would demand the implementation of similar extraction rules from their intermediate representations. However, experimental results from Sawja [11] show that it outperforms Soot in all tests w.r.t. the transformation into their respective intermediate representations, and outperforms Wala w.r.t. virtual method call algorithms. Thus, our extraction algorithm clearly benefits from using Sawja and BIR.

4 Correctness of CFG Extraction

This section discusses the correctness proof of the CFG extraction algorithm. Providing a direct proof for our two-phase extraction is cumbersome. Instead, we prove correctness indirectly, using as reference an idealized *direct extraction algorithm*, denoted $m\mathcal{G}$. The algorithm, defined and proved correct by Amighi [2], is based directly on the semantics of Java bytecode, but assumes an oracle to predict the exceptions that can be thrown by each instruction.

We exploit the idealized algorithm by proving that given a JBC program, the CFG produced by our extraction algorithm ($b\mathcal{G} \circ BC2BIR$) structurally simulates the CFG produced by the direct extraction algorithm ($m\mathcal{G}$). We then reuse an existing result from Gurov *et al.* [10, Th. 36] that structural simulation implies behavioral simulation. By transitivity of simulation we conclude that the behavior induced by the CFG extracted by $b\mathcal{G} \circ BC2BIR$ simulates the JVM behavior. Figure 7 summarizes our approach.

The proof of structural simulation is too large to be presented completely in this paper. Instead, we sketch the overall proof, and discuss one case (for the `throw` instruction) in full detail. For the remaining detailed cases, the reader is referred to the accompanying technical report [3]. Before discussing the proof sketch, we first introduce some terminology and relevant observations.

Preliminaries for the Correctness Proof. The `BC2BIR` transformation may collapse several bytecode instructions into a single BIR instruction. Therefore, we divide bytecode instructions as *producer* instructions, i.e., those that produce at least one BIR instruction in function `BC2BIRinstr`, and *auxiliary* ones, i.e., those that produce none. This division can be deduced from Figure 4 (on page 38).

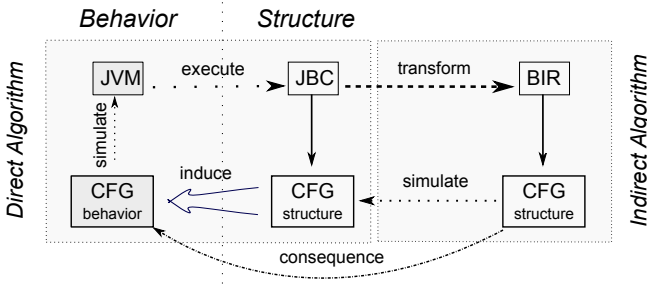


Fig. 7. Schema for CFG extraction and correctness proof

For example, `store` and `invokevirtual` are producer instructions, while `add` and `push` are auxiliary.

We partition the bytecode instruction array into *bytecode segments*. These are subsequences delimited by producer instructions. Thus each bytecode segment contains zero or more contiguous auxiliary instructions, followed by a single producer instruction. Such a partitioning exists for all bytecode programs that comply to the Java bytecode Verifier. All methods in such a program must terminate with `return`, or `athrow`, which are producer instructions. Therefore, there can not be a set of contiguous instructions that is not delimited by a producer instruction.

A *BIR segment* is the result of applying BC2BIR on a bytecode segment. Thus there exists a one-to-one, order-preserving mapping between bytecode segments and BIR segments, and we can associate each JBC or BIR instruction to the unique index of its corresponding bytecode segment.

Figure 5 (on page 39) illustrates the partitioning of instructions into segments. Method `odd` has four bytecode (and BIR) segments, as indicated by the coloring. Producer instructions are underlined.

In the definition of the direct extraction algorithm 3, one can observe that all auxiliary instructions give rise to an internal transfer edge only. This implies that the sub-graphs for any segment extracted in the direct algorithm will start with a path of internal transfer edges with the same size as the number of auxiliary instructions, followed by the edges generated for the producer instruction.

Proof Sketch. Based on observations above, our main theorem states that the method graph extracted using the indirect algorithm weakly simulates (cf. 17) the method graph using the direct algorithm. In the proof, we do not consider the abstract stacks, since only the instructions are relevant to produce the edges.

Theorem 1 (Structural Simulation of Method Graphs). *Let Γ be a well-formed Java bytecode program, and let $\Gamma[m]$ be the implementation of method m . Then $(\text{bG} \circ \text{BC2BIR})(\Gamma[m])$ weakly simulates $\text{mG}(\Gamma[m])$.*

Proof. (Sketch) Let p range over indices in the bytecode instructions array, pc over indices in the BIR instructions array, $\circ_m^{p,x,y}$ over control nodes in

$\mathfrak{mG}(\Gamma[m])$, and $\circ_m^{\text{pc},x,y}$ over control nodes in $(\text{bG} \circ \text{BC2BIR})(\Gamma[m])$. The control nodes are evaluated with two optional atomic propositions: x , which is an exception type, and y , which is the atomic proposition r denoting a return point. Further, let $\text{seg}_{JBC}(m,p)$ and $\text{seg}_{BIR}(m,\text{pc})$ be two auxiliary functions that return the segment number that a bytecode, or a BIR instruction belongs to, respectively, and let function $\text{min}(s,x,y)$ return the least index pc in the BIR segment s resulting in a node valuated with x and y .

We define a binary relation R as follows:

$$R \stackrel{\text{def}}{=} \{ (\circ_m^{p,x,y}, \circ_m^{\text{pc},x,y}) \mid \text{seg}_{JBC}(m,p) = \text{seg}_{BIR}(m,\text{pc}) \wedge \text{pc} = \text{min}(\text{seg}_{BIR}(m,\text{pc}), x, y) \}$$

and show the relation to be a weak simulation in the standard fashion: for every pair of nodes in R , we match every strong transition from the first node by a corresponding weak transition from the second node, so that the target nodes are again related by R . It is easy to establish that the entry nodes of the sub-graphs produced by the two algorithms for the same bytecode segment are related by R , and hence the result.

The proof proceeds by case analysis on the type of the producer instruction of the bytecode segment $\text{seg}_{JBC}(m,p)$. We present one interesting case in full detail; the other cases proceed similarly [3].

*Case **throw*** Let X be the set containing the static type of the exception being thrown, and all of its sub-types. This set is the same for the direct and indirect extraction algorithms. Let $x \in X$.

The direct extraction for the **throw** instruction produces two edges, with the target node of the second edge depending on whether the exception x is caught within the same method it was raised or not (see [3]):

$$\mathfrak{mG}((p, \text{throw}), H) = \begin{cases} \{ \circ_m^p \xrightarrow{\varepsilon} \bullet_m^{p,x}, \bullet_m^{p,x} \xrightarrow{\text{handle}} \circ_m^q \} & \text{if has handler} \\ \{ \circ_m^p \xrightarrow{\varepsilon} \bullet_m^{p,x}, \bullet_m^{p,x} \xrightarrow{\text{handle}} \bullet_m^{p,x,r} \} & \text{otherwise} \end{cases}$$

The transformation $\text{BC2BIR}_{\text{instr}}$ returns a single instruction. Then, similarly to \mathfrak{mG} , the bG function produces two edges (see Figure 6):

$$\begin{aligned} \text{BC2BIR}_{\text{instr}}(p, \text{throw}) &= [\text{throw } x] \\ \text{bG}([\text{throw } x]_{\text{pc}}, H) &= \begin{cases} \{ \circ_m^{\text{pc}} \xrightarrow{\varepsilon} \bullet_m^{\text{pc},x}, \bullet_m^{\text{pc},x} \xrightarrow{\text{handle}} \circ_m^{\text{pc}'} \} & \text{if has handler} \\ \{ \circ_m^{\text{pc}} \xrightarrow{\varepsilon} \bullet_m^{\text{pc},x}, \bullet_m^{\text{pc},x} \xrightarrow{\text{handle}} \bullet_m^{\text{pc},x,r} \} & \text{otherwise} \end{cases} \end{aligned}$$

We have that $(\circ_m^p, \circ_m^{\text{pc}}) \in R$. The transition $\circ_m^p \xrightarrow{\varepsilon} \bullet_m^{p,x}$, is matched by the corresponding weak transition $\circ_m^{\text{pc}} \Longrightarrow \bullet_m^{\text{pc},x}$. Thus obviously also $(\bullet_m^{p,x}, \bullet_m^{\text{pc},x}) \in R$. Next, there are two possibilities for the remaining transitions, depending on whether there is an exception handler for x in p and pc . If there is a handler, then we get $\bullet_m^{p,x} \xrightarrow{\text{handle}} \circ_m^q$, $\bullet_m^{\text{pc},x} \xrightarrow{\text{handle}} \circ_m^{\text{pc}'}$, and clearly also $(\circ_m^q, \circ_m^{\text{pc}'}) \in R$. If there is no exception handler for x , we get $\bullet_m^{p,x} \xrightarrow{\text{handle}} \bullet_m^{p,x,r}$, $\bullet_m^{\text{pc},x} \xrightarrow{\text{handle}} \bullet_m^{\text{pc},x,r}$, and also $(\bullet_m^{p,x,r}, \bullet_m^{\text{pc},x,r}) \in R$. This concludes the case. \square

5 Related Work

Java bytecode has several aspects of an object-oriented language that make the extraction of control-flow graphs complex, such as inheritance, exceptions, and virtual method calls. Therefore, in this section we discuss the work related to extracting CFGs from object-oriented languages. To the best of our knowledge, for none of the existing extraction algorithms a correctness proof has been provided.

Sinha *et al.* [18,19] propose a control-flow graph extraction algorithm for both Java source and bytecode, which takes into account *explicit* exceptions only. The algorithm performs first an intra-procedural analysis, computing the exceptional return nodes caused by uncaught exceptions. Next, it executes an inter-procedural analysis to compute exception propagation paths. This division is similar to how our algorithm analyses exceptional flows, using a slightly different inter-procedural analysis. However, the authors do not discuss how the static type of explicit exceptions is determined by the bytecode analysis, whereas we get this information from the BIR transformation. Moreover, the use of BIR allows us to also support (a subset of the) *implicit* exceptions.

Jiang *et al.* [15] extend the work of Sinha *et al.* to C++ source code. C++ has the same scheme of `try-catch` and exception propagation as Java, but without the `finally` blocks, or implicit exceptions. This work does not consider the exceptions types. Thus, it heavily over-approximates the possible flows by connecting the control points with explicit `throw` within a `try` block to all its `catch` blocks, and considering that any called method containing a `throw` may terminate exceptionally. Our work consider the exceptions types. Thus, it produces more refined CFGs, and also tells which exceptions can be raised, or propagated from method invocations.

Choi *et al.* [8] use an intermediate representation from the Jalapeño compiler [7] to extract CFGs with exceptional flows. The authors introduce a stack-less representation, using assertions to mark the possibility of an instruction raising an exception. This approach was followed by Demange *et al.* when defining BIR, and proving the correctness of the transformation from bytecode. As a result, our extraction algorithm, via BIR, is very similar to that of Choi. We differ by defining formal extraction rules, and proving its correctness w.r.t. behavior.

Finally, Jo and Chang [16] construct CFGs from Java source code by computing normal and exceptional flows separately. An iterative fixed-point computation is then used to merge the exceptional and the normal control-flow graphs. Our exception propagation computation follows their approach; however, the authors do not discuss how the exception type is determined. Also, only explicit exceptions are supported; in contrast, we determine the exception type and support implicit exceptions by using the BIR transformation.

6 Conclusion

This paper presents an efficient and sound control-flow graph extraction algorithm from Java bytecode that takes into account exceptional control flow. The

extracted CFGs can be used for various control-flow analyses, in particular model checking. The algorithm is precise because it is based on BIR, an intermediate stack-less bytecode representation, which provides precise information about exceptional control-flow, and the result is more compact than the original bytecode.

The algorithm is presented formally as an extraction function. We state and prove its soundness: the behavior of the extracted graphs is shown to over-approximate the behavior of the original programs. To the best of our knowledge, this is the first CFG extraction algorithm that has been proved correct. The proof is non-trivial, relying on several results to obtain a relatively economic correctness argument phrased in terms of structural simulation. We believe that the proposed proof strategy, with the level of detail we provide, paves the ground for a mechanized proof using a standard theorem prover.

The extraction algorithm is implemented as the CONFLEX tool. The experimental results confirm that the algorithm is efficient, and that it produces compact CFGs.

Future Work. The extraction algorithm has been designed with modularity in mind. Currently, we investigate how to relativize the algorithm on interface specifications of program modules in order to support modular control-flow graph extraction. In particular, we target CVPP (see *e.g.* [20,13]), a framework and tool set for compositional verification of control-flow safety properties. In this setting, one typically wishes to produce CFGs from incomplete programs.

In addition, we will study how to adapt the algorithm to various generalizations of the program model, including data and multi-threading [12], and how to customize it for other types of instructions (besides method calls and exceptions).

Acknowledgments. We thank the Celtique team at INRIA Rennes for their clarifications about BIR and Sawja. Amighi and Huisman are partially supported by ERC grant 258405 for the VerCors project.

References

1. ConFLEX, <http://www.csc.kth.se/~pedrodcg/conflex>
2. Amighi, A.: Flow Graph Extraction for Modular Verification of Java Programs. Master's thesis, KTH Royal Institute of Technology, Stockholm, Sweden (February 2011), http://www.nada.kth.se/utbildning/grukth/exjobb/rapportlistor/2011/rapporter11/amighi_afshin_11038.pdf, Ref.: TRITA-CSC-E 2011:038
3. Amighi, A., de Carvalho Gomes, P., Gurov, D., Huisman, M.: Provably correct control-flow graphs from Java programs with exceptions. Tech. rep., KTH Royal Institute of Technology (2012), <http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-61188>
4. Bacon, D.F., Sweeney, P.F.: Fast static analysis of C++ virtual function calls. In: OOPSLA, pp. 324–341 (1996)
5. Barre, N., Demange, D., Hubert, L., Monfort, V., Pichardie, D.: SAWJA API documentation (June 2011), <http://javalib.gforge.inria.fr/doc/sawja-api/sawja-1.3-doc/api/index.html>

6. Besson, F., Jensen, T., Le Métayer, D., Thorn, T.: Model checking security properties of control flow graphs. *J. of Computer Security* 9(3), 217–250 (2001)
7. Burke, M.G., Choi, J.D., Fink, S., Grove, D., Hind, M., Sarkar, V., Serrano, M.J., Sreedhar, V.C., Srinivasan, H., Whaley, J.: The Jalapeño dynamic optimizing compiler for Java. In: *Proceedings of the ACM 1999 conference on Java Grande, JAVA 1999*, pp. 129–141. ACM, New York (1999)
8. Choi, J.D., Grove, D., Hind, M., Sarkar, V.: Efficient and precise modeling of exceptions for the analysis of Java programs. *SIGSOFT Softw. Eng. Notes* 24, 21–31 (1999)
9. Demange, D., Jensen, T., Pichardie, D.: A provably correct stackless intermediate representation for Java bytecode. Tech. Rep. 7021, Inria Rennes (2009), <http://www.irisa.fr/celtique/demange/bir/rr7021-3.pdf>, version 3 (November 2010)
10. Gurov, D., Huisman, M., Sprenger, C.: Compositional verification of sequential programs with procedures. *Information and Computation* 206(7), 840–868 (2008)
11. Hubert, L., Barré, N., Besson, F., Demange, D., Jensen, T., Monfort, V., Pichardie, D., Turpin, T.: Sawja: Static Analysis Workshop for Java. In: Beckert, B., Marché, C. (eds.) *FoVeOOS 2010. LNCS*, vol. 6528, pp. 92–106. Springer, Heidelberg (2011)
12. Huisman, M., Aktug, I., Gurov, D.: Program Models for Compositional Verification. In: Liu, S., Araki, K. (eds.) *ICFEM 2008. LNCS*, vol. 5256, pp. 147–166. Springer, Heidelberg (2008)
13. Huisman, M., Gurov, D.: CVPP: A Tool Set for Compositional Verification of Control-Flow Safety Properties. In: Beckert, B., Marché, C. (eds.) *FoVeOOS 2010. LNCS*, vol. 6528, pp. 107–121. Springer, Heidelberg (2011)
14. IBM: T.J. Watson Libraries for Analysis (Wala). <http://wala.sourceforge.net/>
15. Jiang, S., Jiang, Y.: An analysis approach for testing exception handling programs. *SIGPLAN Not.* 42, 3–8 (2007)
16. Jo, J.-W., Chang, B.-M.: Constructing Control Flow Graph for Java by Decoupling Exception Flow from Normal Flow. In: Laganá, A., Gavrilova, M.L., Kumar, V., Mun, Y., Tan, C.J.K., Gervasi, O. (eds.) *ICCSA 2004. LNCS*, vol. 3043, pp. 106–113. Springer, Heidelberg (2004)
17. Milner, R.: *Communicating and mobile systems: the π -calculus*, ch. 6, pp. 52–53. Cambridge University Press, New York (1999)
18. Sinha, S., Harrold, M.J.: Criteria for testing exception-handling constructs in Java programs. In: *Proceedings of the IEEE International Conference on Software Maintenance, ICSM 1999*, pp. 265–276. IEEE Computer Society (1999)
19. Sinha, S., Harrold, M.J.: Analysis and testing of programs with exception handling constructs. *IEEE Trans. Softw. Eng.* 26, 849–871 (2000)
20. Soleimanifard, S., Gurov, D., Huisman, M.: ProMoVer: Modular Verification of Temporal Safety Properties. In: Barthe, G., Pardo, A., Schneider, G. (eds.) *SEFM 2011. LNCS*, vol. 7041, pp. 366–381. Springer, Heidelberg (2011)
21. Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., Godin, C.: Practical virtual method call resolution for java. In: *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2000*, pp. 264–280. ACM, New York (2000), <http://doi.acm.org/10.1145/353171.353189>
22. Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java Optimization Framework. In: *CASCON 1999*, pp. 125–135 (1999), <http://www.sable.mcgill.ca/soot/>

Checking Sanity of Software Requirements^{*}

Jiří Barnat, Petr Bauch, and Luboš Brim

Faculty of Informatics, Masaryk University
Brno, Czech Republic

{barnat, xbauch, brim}@fi.muni.cz

Abstract. In the last decade it became a common practice to formalise software requirements to improve the clarity of users' expectations. In this work we build on the fact that functional requirements can be expressed in temporal logic and we propose new techniques that automatically detect flaws and suggest improvements of given requirements. Specifically, we describe and experimentally evaluate new approaches to consistency and vacuity checking that identify all inconsistencies and pinpoint their exact source (the smallest inconsistent set). To complete the sanity checking we also deliver a novel semi-automatic completeness evaluation that can assess the coverage of user requirements and suggest missing properties the user might have wanted to formulate. The usefulness of our completeness evaluation is demonstrated in a case study of an aeroplane control system.

1 Introduction

The earliest stages of software development entail among others the activity of user requirements elicitation. The importance of clear specification of the requirements in the contract-based development process is apparent from the necessity of final-product compliance verification. Yet the specification itself is rarely described formally. Nevertheless, the formal description is an essential requirement for any kind of comprehensive verification. Recently, there have been tendencies to use the mathematical language of temporal logics, e.g. the *Linear Temporal Logic* (LTL), to specify functional system requirements. Restating requirements in a rigorous, formal way enables the requirement engineers to scrutinise their insight into the problem and allows for a considerably more thorough analysis of the final requirement documents [10].

Later in the development, when the requirements are given and a model is designed, the formal verification tools can provide a proof of correctness of the system being developed with respect to formally written requirements. The model of the system or even the system itself can be checked using model checking [11,6] or theorem proving [2] tools. If there are some requirements the system does not meet, the cause has to be found and the development reverted. The longer it takes to discover an error in the development, the more expensive the error is to mend. Consequently, errors made during requirements specification are among the most expensive ones in the whole development.

^{*} This work has been supported by the Czech Grant Agency grant No. GAP202/11/0312 and GD102/09/HD042 and by Artemis-IA iFEST project grant No. 100203.

Model checking is particularly efficient in finding bugs in the design, however, it exhibits some shortcomings when it is applied to requirement analysis. In particular, if the system satisfies the formula then the model checking procedure only provides an affirmative answer and does not elaborate for the reason of the satisfaction. It could be the case that, e.g. the specified formula is a tautology, hence, it is satisfied for any system. To mitigate the situation a subsidiary approach, the sanity checking, was proposed to check vacuity and coverage of requirements [12]. Yet the existence of a model is still a prerequisite which postpones the verification until later phases in the development cycle.

The primary interest of this paper is to design, implement and evaluate techniques that would allow the developers of computer systems to check sanity of their requirements when it matters most, i.e. during the requirements stage. The implementation of our techniques is planned to be incorporated in the iFEST project [11]. Together with a tool translating natural language requirements into LTL formulae our techniques could provide automatic sanity checking procedure of freshly elicited requirements.

Contribution. This paper redefines the notion of sanity checking of requirements written as LTL formulae and describes its implementation and evaluation. The proposed notion liberates sanity checking from the necessity of having a model of the developed system. Sanity checking commonly consists of three parts: consistency and vacuity checking and completeness of requirements. Our approach to consistency and vacuity checking is novel in identifying all inconsistent (or vacuous) subsets of the input set of requirements. This considerably simplifies the work of requirements engineers because it pinpoints all sources of inconsistencies. For completeness checking, we propose a new behaviour-based coverage metric. Assuming that the user specifies what behaviour of the system is sensible, our coverage metric calculates what portion of this behaviour is described by the requirements specifying the system itself. The method further suggests new requirements to the user that would improve the coverage and thus ensure more accurate description of users' expectations. The efficiency and usability of our approach to sanity checking is verified in an experimental evaluation and a case study.

1.1 Related Work

The use of model checking with properties (specified in CTL) derived from real-life avionics software specifications was successfully demonstrated in [3]. This paper intends to present a preliminary to such a use of a model checking tool, because there the authors presupposed sanity of their formulae. The idea of using coverage as a metric for completeness can be traced back to software testing, where it is possible to use LTL requirements as one of the coverage metrics [20,16].

Model-based sanity checking was studied thoroughly and using various approaches, but it is intrinsically different from model-free checking presented in this paper. Completeness is measured using metrics based on the state space coverage of the underlying model [4,5]. Vacuity of temporal formulae was identified as a problem related to model checking and solutions were proposed in [13] and in [12], again expecting existence of a model.

Checking consistency (or satisfiability) of temporal formulae is a well understood problem solved using various techniques in many papers (most recently using SAT-based approach in [17] or in [18] where it was used as a comparison between different LTL to Büchi translation techniques). The classical problem is formulated as to decide whether a set of formulae is internally consistent. In this paper, however, a more elaborate answer is sought: specifically which of the formulae cause the inconsistency. The approach is then extended to vacuity which is rarely used in model-free sanity checking.

Completeness of formal requirements is not as well-established and its definition often differs. The most thorough research in algorithmic evaluation of completeness was conducted in [9,14,15]. Authors of those papers use RSML (Requirements State Machine Language) to specify requirements which they translate (in the last paper) to CTL and to a theorem prover language to discover inconsistencies and detect incompleteness. Their notion of completeness is based on verifying that for every input value there is a reaction described in the specification. This paper presents completeness as considering all behaviours described as sensible (and either refuting or requiring them). Finally, a novel semi-formal methodology is proposed in this paper, that recommends new requirements to the user, that have the potential to improve completeness of the input specification.

2 Preliminaries

This section serves as a motivation for and a reminder of the model checking process and its connection to sanity checking. A knowledgeable reader might find it slow-paced and cursory, but its primary function is to justify the use of formal specifications and, as such, requires more compliant approach.

2.1 LTL Model Checking

Definition 1. Let AP be the set of atomic propositions. Then this recursive definition specifies all well-formed LTL formulae over AP , where $p \in AP$:

$$\Psi ::= p \mid \neg\Psi \mid \Psi \wedge \Psi \mid X \Psi \mid \Psi U \Psi$$

Example 1. There are some well-established syntactic simplifications of the LTL language, e.g. $false := p \wedge \neg p$, $true := \neg false$, $\phi \Rightarrow \psi := \neg(\phi \wedge \neg\psi)$, $F \phi := true U \phi$, $G \phi := \neg(F \neg\phi)$. Assuming that $AP = \{\alpha := (c = 5), \beta := (a \neq b)\}$, these are examples of well-formed LTL formulae: $G \beta, \alpha U \neg\beta$.

In classical model checking one usually verifies that a model of the system in question satisfies the given set of LTL-specified requirements. That is not possible in the context of this paper because in the requirements stage there is no model to work with. Nevertheless, to better understand the background of LTL model checking let us assume that the system is modelled as a *Labelled Transition System* (LTS).

Definition 2. Let Σ be a set of state labels (it will mostly hold that $\Sigma = AP$). Then an LTS $M = (S, \rightarrow, \nu, s_0)$ is a tuple, where: S is a set of states, $\rightarrow \subseteq S \times S$ is a transition

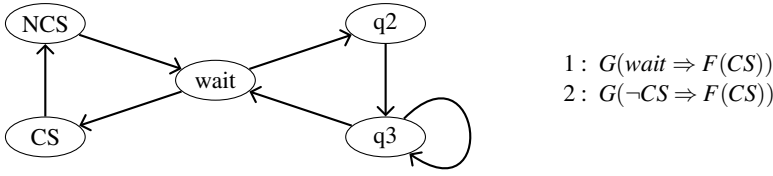


Fig. 1. LTS for Peterson's mutual exclusion protocol (with only one process) and two liveness LTL formulae

relation, $v : S \rightarrow 2^\Sigma$ is a valuation function and $s_0 \in S$ is the initial state. A function $r : \mathbb{N} \rightarrow S$ is an infinite run over the states of M if $r(0) = s_0, \forall i : r(i) \rightarrow r(i+1)$. The trace or word of a run is a function $w : \mathbb{N} \rightarrow 2^\Sigma$, where $w(i) = v(r(i))$.

An LTL formula states a property pertaining to an infinite trace (a trace that does not have to be associated with a run). Assuming the LTS is a model of a computer program then a trace represents one specific execution of the program. Also the infiniteness of the executions is not necessarily an error – programs such as operating systems or controlling protocols are not supposed to terminate.

Definition 3. Let w be an infinite word and let Ψ be an LTL formula over Σ . Then it is possible to decide if w satisfies Ψ ($w \models \Psi$) based on the following rules (where $w(i)$ is the i -th letter of w and w_i is the i -th suffix of w):

$$\begin{aligned}
 w \models p & \quad \text{iff } p \in w(0), \\
 w \models \neg \Psi & \quad \text{iff } w \not\models \Psi, \\
 w \models \Psi_1 \wedge \Psi_2 & \quad \text{iff } w \models \Psi_1 \text{ and } w \models \Psi_2, \\
 w \models X \Psi & \quad \text{iff } w_1 \models \Psi, \\
 w \models \Psi_1 U \Psi_2 & \quad \text{iff } \exists i \forall j < i : w_j \models \Psi_1, w_i \models \Psi_2,
 \end{aligned}$$

Example 2. Figure 1 contains LTS for a process engaged in Peterson's mutual exclusion protocol. The protocol can control access to the critical section (state CS) for arbitrarily many processes that communicate using global variables to determine which process will be granted access next. The two LTL formulae verify the liveness property of the protocol, e.g. 1: if a process *waits* for CS then it will eventually get there.

Clearly, a system as a whole satisfies an LTL formula if all its executions (all infinite words over the states of its LTS) do. Efficient verification of that satisfaction, however, requires a more systematic approach than enumeration of all executions. An example of a successful approach is the enumerative approach using *Büchi automata*.

Definition 4. A Büchi automaton is a pair $A = (M, F)$, where M is an LTS and $F \subseteq S$. An automaton A accepts an infinite word w ($w \in L(A)$) if there exists a run r for w in M and there is a state from F that appears infinitely often on r , i.e. $\forall i \exists j > i : r(j) \in F$.

Arbitrary LTL formula ϕ can be transformed into a Büchi automaton A_ϕ such that $w \models \phi \Leftrightarrow w \in L(A_\phi)$. Also checking that every execution satisfies ϕ is equivalent to checking that no execution satisfies $\neg\phi$. It only remains to combine the LTS model of

the given system M with $A_{\neg\phi}$ in such a way that the resulting automaton will accept exactly those words of M that violate ϕ . Finally, deciding existence of such a word – and by extension verifying correctness of the system – has been shown equivalent to finding accepting cycle in a graph.

2.2 Model-Based Sanity Checking

As described in the introduction the model checking procedure is not designed to decide why was a certain property satisfied in a given system. That is a problem, however, because the reasons for satisfaction might be the wrong ones. If for example the system is modelled erroneously or the formula is not appropriate for the system then it is still possible to receive a positive answer.

These kinds of inconsistencies between the model and the formula are detected using sanity checking techniques, namely *vacuity* and *coverage*. In this paper they will be described only for comparison with their model-free versions and interested reader should consult for example [12] for more details.

Let K be an LTS, ϕ a formula and ψ its subformula. Then ψ *does not affect the truth value of ϕ in K* if K satisfies $\forall x\phi[\psi/x]$ if and only if K satisfies ϕ , where $[\psi/x]$ substitutes x with ψ . Then a system K *satisfies a formula ϕ vacuously* iff $K \models \phi$ and there is a subformula ψ of ϕ such that ψ does not affect ϕ in K .

A state s of an LTS K is *q -covered by ϕ* , for a formula ϕ and an atomic proposition q , if K satisfies ϕ but $\tilde{K}_{s,q}$ does not satisfy ϕ . There $\tilde{K}_{s,q}$ stands for an LTS equivalent to K except the valuation of q in the state s is flipped.

It can be observed that these notions of sanity are deeply dependent on the system that is being verified. In order to be used without a model these notions need to be altered considerably while preserving the main idea. Vacuity states that the satisfaction of a formula is given extrinsically and is not related to the formula itself. Coverage, on the other hand, attempts to capture the amount of system behaviour that is described by the formulae. These concepts are in this paper translated into model-free environment and supplemented with consistency verification to form a complex sanity checking.

3 Model-Free Sanity Checking

As various studies concluded, undetected errors made early in the development cycle are the most expensive to eradicate. Thus it is very important that the outcome of the requirements stage – a database of well-formed, traceable requirements – is what the customer intended and that nothing was omitted (not even unintentionally). While a procedure that would ensure the two properties cannot be automated, this paper proposes a methodology to check the sanity of requirements. In the following the *sanity checking* will be considered to consist of 3 related tasks: *consistency*, *vacuity* and *completeness* checking.

Definition 5. A set Γ of LTL formulae over AP is consistent if $\exists w \in AP^\omega : w$ satisfies $\bigwedge \Gamma$. Checking consistency of a set Γ entails finding all minimal inconsistent subsets of Γ . A formula ϕ is vacuous with respect to a set of formulae Γ if $\bigwedge \Gamma \Rightarrow \phi$. To check vacuity of a set Γ entails finding all pairs of $\langle \phi \in \Gamma, \Phi \subseteq \Gamma \rangle$ such that Φ is consistent and $\Phi \Rightarrow \phi$ (and for no $\Phi' \subseteq \Phi$ does it hold that $\Phi' \Rightarrow \phi$).

The existence of the appropriate w can be tested by constructing $A_{\wedge\Gamma}$ and checking that $L(A_{\wedge\Gamma})$ is non-empty. The procedure is effectively equivalent to model checking where the model is a clique over the graph with one vertex for every element of 2^{A^P} (allowing every possible behaviour).

This approach to consistency and vacuity is especially efficient if a large set of requirements needs to be processed and standard sanity checking would only reveal if there is an inconsistency (or vacuity) but would not be able to locate the source. Furthermore, dealing with larger sets of requirements entails the possibility that there will be several inconsistent subsets or that a formula is vacuous due to multiple small subsets. Each of these conflicting subsets needs to be considered separately which can be facilitated using the methodology proposed in this paper.

Example 3. Let us assume that there are five requirements formalised as LTL formulae over a set of atomic propositions $\{p, q, a\}$. They are $1 : F(p \Rightarrow p U q)$, $2 : GF(p)$, $3 : G\neg(a \wedge p)$, $4 : G(X q \Rightarrow a)$ and $5 : GF(q)$. In this set the formula 4 is inconsistent due to the first 3 formulae and the last formula is vacuously satisfied (implied) by the first 2 formulae.

3.1 Implementation of Sanity Checking

Let us henceforth denote one specific instance of consistency (or vacuity) checking as a *check*. For consistency and a set Γ it means to check that for some $\gamma \subseteq \Gamma$ is $\bigwedge \gamma$ satisfiable. For vacuity it means for $\gamma \subseteq \Gamma$ and $\phi \in \Gamma$ to check that $\bigwedge \gamma \Rightarrow \phi$ is satisfiable. In the worst case both consistency and vacuity checking would require an exponential number of checks. However, the proposed algorithm considers previous results and only performs the checks that need to be tested.

Both consistency and vacuity checking use three data structures that facilitate the computation. First there is the queue of verification tasks called *Pool*, then there are two sets, *Con* and *Incon*, which store the consistent and inconsistent combinations found so far. Finally, each individual *task* contains a set of integers (that uniquely identifies formulae from Γ) and a *flag* value (containing three bits for three binary properties). First, whether the satisfaction check was already performed or not. Second, if the combination is consistent. And the third bit specifies the direction in subset relation (up or down in the Hasse diagram) in which the algorithm will continue. The successors will be either subsets or supersets of the current combination.

The idea behind consistency checking is very simple (listed as Algorithm [1](#)). The pool contains all the tasks to be performed and these tasks are of two types: either to check consistency of the combination or to generate successors. The symmetry of the solution allows for parallel processing (multiple threads performing the Algorithm [1](#) at the same time) given that the data structures are properly protected from race conditions. The pool needs to be initialised with all single element subsets of Γ and Γ itself, thus in the subsequent iteration will be checked the supersets of the former and subsets of the latter.

Algorithm [2](#) is called when the task t on the top of *Pool* is already checked. At this point either all subsets or all supersets of t should be enqueued as tasks. But not all successors need to be inspected, e.g. if t is consistent then also all its subsets will be consistent – that is clearly true and no subset of t needs to be checked.

| | |
|--|---|
| <p>Algorithm 1. Consistency Check</p> <pre> 1 while t ← getTask() do 2 if t.checked() then 3 genSuccs(t) 4 else 5 t.checked ← verCons(t) 6 updateSets(t) 7 Pool.enqueue(t) </pre> | <p>Algorithm 2. genSuccs(<i>Task</i> t)</p> <pre> 1 if t.con() then 2 if t.dir = ↑ then 3 genSupsets(t) 4 else 5 if t.dir = ↓ then 6 genSubsets.(t) </pre> |
| <p>Algorithm 3. genSupsets(<i>Task</i> t)</p> <pre> 1 foreach i ∈ {1, ..., n} do 2 t.add(i) 3 if ∀X ∈ Con : X ⊈ t ∧ 4 ∀X ∈ Incon : X ⊆ t then 5 enqueue(t) 6 t.erase(i) </pre> | <p>Algorithm 4. verCons($t = \langle i_1, \dots, i_j \rangle$)</p> <pre> 1 F ← createConj($\phi_{i_1}, \dots, \phi_{i_j}$) 2 A ← transform2BA(F) 3 return findAccCycle(A) </pre> |

That observation is utilised again in Algorithm 3. It does not suffice to stop generating subsets and supersets when its immediate predecessors are found consistent (inconsistent), because it can also happen that the combination to be checked was formed in a different branch of the Hasse diagram of the subset relation. In order to prevent redundant satisfiability checks two sets are maintained *Con* and *Incon* (see how these are used on line 4 of Algorithm 3).

The actual consistency (and quite similarly also vacuity) checking is less complicated (see Algorithm 4). First, the conjunction of formulae encoded in the task is created, then the appropriate Büchi automaton is checked for existence of an accepting cycle: using nested DFS [7]. The only difference when performing the vacuity checking is that the task *t* consists of a list $\langle i_1, \dots, i_j \rangle$ which can be empty, and one index i_k . Since the task is to check that $\phi_{i_1} \wedge \dots \wedge \phi_{i_j} \Rightarrow \phi_{i_k}$ the line 1 needs to be altered to $F \leftarrow \text{createConj}(\phi_{i_1}, \dots, \phi_{i_j}, \neg\phi_{i_k})$. Because if $\phi_{i_1} \wedge \dots \wedge \phi_{i_j} \wedge \neg\phi_{i_k}$ is satisfiable then $\phi_{i_1} \wedge \dots \wedge \phi_{i_j} \not\Rightarrow \phi_{i_k}$, i.e. ϕ_{i_k} is not satisfied vacuously by $\{\phi_{i_1}, \dots, \phi_{i_j}\}$.

Discarding the Büchi automata in every iteration may seem unnecessarily wasteful, especially since synchronous composition of two (and more) automata is a feasible operation. However, the size of an automaton created by composition is a multiplication of the sizes of the automata being composed. Furthermore, it would not be possible to use the size optimising techniques employed in LTL to Büchi translation. And these techniques work particularly well in our case, because the translated formulae (conjunctions of requirements) have relatively small nesting depth (maximal depth among requirements + 1).

4 Completeness Checking

The completeness checking is a little more involved: this is in fact the part that provably cannot be fully automated. Hence the paper will first describe the problem and then detail the semi-automatic solution proposed.

Let us assume that the user specifies three types of requirements: environmental assumptions Γ_A , required behaviour Γ_R and forbidden behaviour Γ_F . The environmental assumptions represent the sensible properties of the world a given system is to work in, e.g. “*The plane is either in the air or on the ground, but never in both these states at once*”. The required behaviour represents the functional requirements imposed on the system: the system will not be correct if any of these is not satisfied. Dually, the forbidden behaviour contains those patterns that the system must not display. Assume henceforth the following simplifying notation for Büchi automata: let f be a propositional formula over capital Latin letters A, B, \dots barred letters \bar{A}, \bar{B}, \dots and Greek letters α, β, \dots , where A substitutes $\bigwedge_{\gamma \in \Gamma_A} \gamma$, \bar{A} stands for $\bigvee_{\gamma \in \Gamma_A} \gamma$ and all Greek letters represent simple LTL formulae. Then \mathcal{A}_f denotes such a Büchi automaton that accepts all words satisfying the substituted f , e.g. $\mathcal{A}_{A \vee \bar{B} \wedge \phi}$ accepts words satisfying $\bigwedge_{\gamma \in \Gamma_A} \gamma \vee \bigvee_{\gamma \in \Gamma_B} \gamma \wedge \phi$. The automaton \mathcal{A}_A thus describes the part of the state space the user is interested in and which the required and forbidden behaviour should together submerge. That most commonly is not the case with freshly elicited requirements and therefore the problem is the following: find sufficiently simple formulae over AP that would, together with formulae for R and F , cover a large portion of \mathcal{A}_A . In other words to find such ϕ that $\mathcal{A}_{R \vee \bar{F} \vee \phi}$ covers as much of \mathcal{A}_A as possible.

In order to evaluate the size of the part of \mathcal{A}_A covered by a single formula, i.e. how *much* of the possible behaviour is described by it, an evaluation methodology for Büchi automata needs to be established. The plain enumeration of all possible words accepted by an automaton is impractical given the fact that Büchi automata operate over infinite words. Similarly, the standard completeness metrics based on state coverage [5][19] are unsuitable because they do not allow for comparison of sets of formulae and they require the underlying model. Equally inappropriate is to inspect only the underlying directed graph because Büchi automata for different formulae may have isomorphic underlying graphs.

The methodology proposed in this paper is based on the notion of *almost-simple* paths and *directed partial coverage* function.

Definition 6. Let G be a directed graph. A path π in G is a sequence of vertices v_1, \dots, v_n such that $\forall i : (v_i, v_{i+1})$ is an edge in G . A path is almost-simple if no vertex appears on the path more than twice. The notion of almost-simplicity is also applicable to words in the case of Büchi automata.

With almost-simple paths one can enumerate the behavioural patterns of a Büchi automaton without having to incorporate infinity. Clearly, it is a heuristic approach and a considerable amount of information will be lost but since all simple cycles will be considered the resulting evaluation should provide sufficient distinguishing capacity (as demonstrated in Section 5.2).

Knowing which paths are interesting it is possible to propose a methodology that would allow comparing two paths. There is, however, a difference between Büchi automata that represent a computer system and those built using only LTL formulae (that will restrict the behaviour of the former). The latter automata use a different evaluation function \hat{v} that assigns to every edge a set of literals. The reason behind this is that the LTL-based automaton only allows those edges (in the system automaton) for which their source vertex has evaluation compatible with the edge evaluation (now in the LTL automaton).

Definition 7. Let \mathcal{A}_1 and \mathcal{A}_2 be two (LTL) Büchi automata over AP and let AP_L be the set of literals over AP. The directed partial coverage function Λ assigns to every pair of edge evaluations a rational number between 0 and 1, $\Lambda : 2^{AP_L} \times 2^{AP_L} \rightarrow \mathbb{Q}$. The evaluation works as follows (where $p = |A_1 \cap A_2|$):

$$\Lambda(A_1 = \{l_{11}, \dots, l_{1n}\}, A_2 = \{l_{21}, \dots, l_{2m}\}) = \begin{cases} 0 & \exists i, j : l_{1i} \equiv \neg l_{2j} \\ p/m & \text{otherwise} \end{cases}$$

From the definition one can observe that Λ is not symmetric. This is intentional because the goal is to evaluate how much a path in \mathcal{A}_2 covers a path in \mathcal{A}_1 . Hence the fact that there are some additional restricting literals on an edge of \mathcal{A}_1 does not prevent automaton \mathcal{A}_2 to display the required behaviour (the one observed in \mathcal{A}_1).

The extension of coverage from edges to paths and automata is based on averaging over almost-simple paths. An almost-simple path π_2 of automaton \mathcal{A}_2 covers an almost-simple path π_1 of automaton \mathcal{A}_1 by $\Lambda(\pi_1, \pi_2) = \frac{\sum_{i=0}^n \Lambda(A_{1i}, A_{2i})}{n}$ per cent, where n is the number of edges and A_{ji} is the set of labels on i -th edge on π_j . Then automaton \mathcal{A}_2 covers \mathcal{A}_1 by $\Lambda(\mathcal{A}_1, \mathcal{A}_2) = \frac{\sum_{i=0}^m \max_{\pi_1} \Lambda(\pi_1, \pi_{2i})}{m}$ per cent, where m is the number of almost-simple paths of \mathcal{A}_1 that end in an accepting vertex. It follows that coverage of 100 per cent occurs when for every almost-simple path of one automaton there is an almost-simple path in the other automaton that exhibits similar behaviour.

4.1 Implementation of Completeness Checking

The high-level overview of the implementation of the proposed methodology is based on partial coverage of almost-simple paths of \mathcal{A}_A . In other words finding the most suitable path in $\mathcal{A}_{R \vee \bar{F} \vee \phi}$ for every almost-simple path in \mathcal{A}_A , where ϕ is a sensible simple LTL formula that is proposed as a candidate for completion. Finally, the suitability will be assessed as the average of partial coverage over all edges on the path.

The output of such a procedure will be a sequence of candidate formulae, each associated with an estimated coverage (a number between 0 and 1) the addition of this particular formula would entail. The candidate formulae are added in a sequence so that the best unused formula is selected in every round. Finally, the coverage is always related to \mathcal{A}_A and, thus, if some behaviour that cannot be observed in \mathcal{A}_A is added with a candidate formula this addition will neither improve nor degrade the coverage.

Example 4. The method of Büchi automata evaluation will be partially exemplified using Figure 2. The example only shows the enumeration of almost-simple paths and

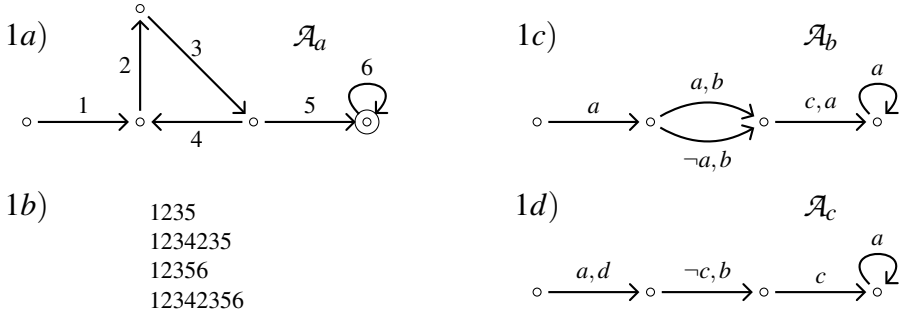


Fig. 2. 1a) Example Büchi automaton \mathcal{A}_a ; 1b) All almost-simple paths of \mathcal{A}_a ; 1c) and 1d) are two different Büchi automata with relatively similar evaluations of almost-simple paths (see Example 4)

the partial coverage of two paths. What remains to the complete methodology will be shown more structurally in Algorithm 5. The enumeration of almost-simple paths of \mathcal{A}_a in Figure 1a) should be straightforward, part the fact that a path is represented as a sequence of edges for simplicity. Let us assume that \mathcal{A}_b is the original automaton and \mathcal{A}_c is being evaluated for how thoroughly it covers \mathcal{A}_b . There are 4 almost-simple paths in \mathcal{A}_b , one of them is $\pi = \langle a; -a, b; c, a; a \rangle$. The partial coverage between the first edge of π and the first edge of \mathcal{A}_c (there is only one possibility) is 0.5, since there is the excessive literal d . The coverage between the second edges is also 0.5, but only because of $\neg c$ in \mathcal{A}_c ; the superfluous literal $\neg a$ restricts only the behaviour of \mathcal{A}_b . Finally, the average similarity between π and the respective path in \mathcal{A}_c is 0.75 and it is approximately 0.7 between the two automata.

The topmost level of the completeness evaluation methodology is shown as Algorithm 5. As input this function requires the three sets of user defined requirements, the set of candidate formulae and the number of formulae the algorithm needs to select. On lines 1 and 2 the formulae for conjunction of assumptions and user requirements (both required and forbidden) are created. They will be used later to form larger formulae to be translated into Büchi automata and evaluated for completeness but, for now, they need to be kept separate. Next step is to enumerate the almost-simple paths of \mathcal{A}_A for later comparison, i.e. a baseline state space that the formulae from Γ_{Cand} should cover.

The rest of the algorithm forms a cycle that iteratively evaluates all candidates from Γ_{Cand} (see line 8 where the corresponding formula is being formed). Among the candidate formulae the one with the best coverage of the paths is selected and subsequently added to the covering system.

Functions `enumeratePaths` and `avrPathCov` are similar extensions of the BFS algorithms. Unlike BFS, however, they do not keep the set of visited vertices to allow state revisiting (twice in case of `enumeratePaths` and arbitrary number of times in case of `avrPathCov`). The `avrPathCov` search is executed once for every path it receives as input and stops after inspecting all paths to the length of the input path or if the current search path is incompatible (see Definition 7).

Algorithm 5. Completeness Evaluation

Input : $\Gamma_A, \Gamma_R, \Gamma_F, \Gamma_{Cand}, n$
Output: Best coverage for $1 \dots n$ formulae from Γ_{Cand}

```

1  $\gamma_{Assum} \leftarrow \bigwedge_{\gamma \in \Gamma_A} \gamma$ 
2  $\gamma_{Desc} \leftarrow \bigwedge_{\gamma \in \Gamma_R} \gamma \vee \bigvee_{\gamma \in \Gamma_F} \gamma$ 
3  $A \leftarrow \text{transform2BA}(\gamma_{Assum})$ 
4  $\text{pathsBA} \leftarrow \text{enumeratePaths}(A)$ 
5 for  $i = 1 \dots n$  do
6    $\text{max} \leftarrow \infty$ 
7   foreach  $\gamma \in \Gamma_{Cand}$  do
8      $\gamma_{Test} \leftarrow \gamma_{Desc} \vee \gamma$ 
9      $A \leftarrow \text{transform2BA}(\gamma_{Test})$ 
10     $\text{cur} \leftarrow \text{avrPathCov}(A, \text{pathsBA})$ 
11    if  $\text{max} < \text{cur}$  then
12       $\text{max} \leftarrow \text{cur}$ 
13       $\gamma_{Max} \leftarrow \gamma$ 
14     $\text{print}(\text{"Best coverage in } i\text{-th round is max."})$ 
15     $\gamma_{Desc} \leftarrow \gamma_{Desc} \vee \gamma_{Max}$ 
16     $\Gamma_{Cand} \leftarrow \Gamma_{Cand} \setminus \{\gamma_{Max}\}$ 

```

5 Experimental Evaluation

All three sanity checking algorithms were implemented as an extension of the parallel explicit-state LTL model checker DiVinE [11]. From the many facilities offered by this tool, only the LTL to Büchi translation was used. As the original tool also its extension was implemented using parallel computation, yet due to space limitation this aspect is not to be described in this paper.

5.1 Experiments with Random Formulae

The first set of experiments was conducted on randomly generated LTL formulae. In order for the experiments to be as realistic as possible formulae with various nesting depths were generated. Nesting depth denotes the depth of the syntactic tree of a formula. Statistics about the most common formulae show, e.g. in [8], that the nesting is rarely higher than 5 and is 3 on average. Following these observations, the generating algorithm takes as input the desired number n of formulae and produces: $n/10$ formulae of nesting 5, $9n/60$ of nesting 1, $n/6$ of nesting 4, $n/4$ of nesting 2 and $n/3$ of nesting 3. Finally, the number of atomic propositions is also chosen according to n (it is $n/3$) so that the formulae would all contribute to the same state space.

All experiments were run on a dedicated Linux workstation with quad core Intel Xeon 5130 @ 2GHz and 16GB RAM. The codes were compiled with optimisation options `-O2` using GCC version 4.3.2. Since the running times and even the number of

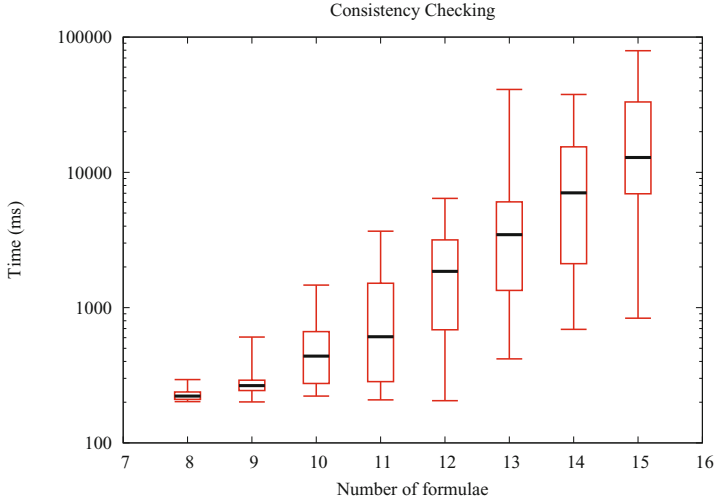


Fig. 3. Log-plot summarising the time complexity of consistency checking

checks needed for completion of all proposed algorithms differ for every set of formulae, the experiments were ran multiple times. The sensible number of formulae starts at 8: for less formulae the running time is negligible. Experimental tests for consistency and vacuity were executed for up to 15 formulae and for each number the experiment was repeated 25 times.

Figure 3 summarises the running times for consistency checking experiments. For every set of experiments (on the same number of formulae) there is one box capturing median, extremes and the quartiles for that set of experiments. From the figure it is clear that despite the optimisation techniques employed in the algorithm both median and maximal running times increase exponentially with the number of formulae. On the other hand there are some cases for which presented optimisations prevented the exponential blow-up as is observable from the minimal running times.

Figure 4 illustrates the discrepancy between the number of combinations of formulae and the number of vacuity checks that were actually performed. The number of combinations for n formulae is $n * 2^{n-1}$ but the optimisation often led to much smaller number. As one can see from the experiments on 9 formulae, it is potentially necessary to check almost all the combinations but the method proposed in this paper requires on average less than 10 per cent of the checks and the relative number decreases with the number of formulae.

5.2 Case Study: Aeroplane Control System

A sensible exposition of the effectivity of completeness evaluation proposed in this paper requires more elaborate approach than using random formulae. For that purpose a case study has been devised that demonstrates the capacity to assess coverage of requirements and to recommend suitable coverage-improving LTL formulae. Random

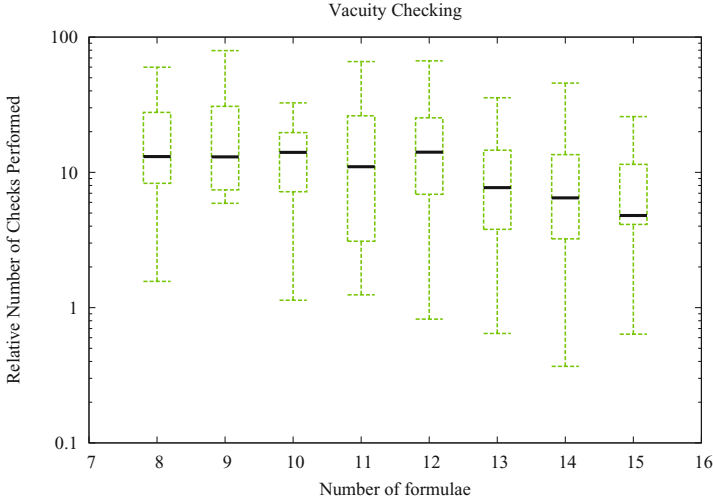


Fig. 4. Log-plot with the relative number of checks for vacuity checking

formulae were used only as candidates for coverage improvement: they were built based on the atomic proposition that appeared in the input formulae and only very simple generated formulae were selected. It was also required that the candidates do not form an inconsistent or tautological set. Alternatively, pattern-based [8] formulae could be used as candidates. The methodology is general enough to allow semi-random candidates generated using patterns from input formulae. For example if an implication is used as the input formula, the antecedent may not be required by the other formulae which may not be according to user's expectations.

The case study attempts to propose a system of LTL formulae that should control the flight and more specifically the landing of an aeroplane. The LTL formulae and the atomic propositions they use are summarised in Figure 5. The requirements are divided into 3 categories similarly as in the text: *A* requirements represent assumptions and *R* and *F* stand for required and forbidden behaviour. For example the formula *R2* expresses the requirement that whenever *landing* the plane should eventually slow down from 200 mph to 100 mph (during which the speed never goes above 200 mph).

Initially, the coverage of *R* and *F* requirements is 0. Though they were not selected specifically for the purpose of covering the *A* requirements, it is still alarming that not a single path was preserved. The first formula selected by the Algorithm 5 and leading to coverage of 9.1 per cent was a simple $G(\neg l)$. Not particularly interesting per se, nonetheless emphasising the fact that without this requirement, landing was never required. Unlike the previous formula which would be added to forbidden behaviour, the next select formula ($F(\neg b \wedge \neg l)$) is clearly required totalling the coverage to 39.4 per cent. This formula points out that the required behaviour only specifies what should happen after landing, unlike assumption which also require flight. The final formula $F(\neg l U (a \vee b))$ connects flight and landing and its addition entails coverage of 54.9 per cent.

Atomic Propositions

$a \equiv [\text{height} = 0]$
 $b \equiv [\text{speed} \leq 200]$
 $l \equiv [\text{landing}]$
 $u \equiv [\text{undercarriage}]$
 $c \equiv [\text{speed} \leq 100]$

LTL Requirements

$A1 : G(a \Leftrightarrow b)$ $R1 : G(l \Rightarrow F(G(b) \wedge F(G(c))))$
 $A2 : F(l) \wedge G(l \Rightarrow F(a))$ $R2 : G(l \Rightarrow F(b \ U \ c))$
 $A3 : G(\neg l \Rightarrow \neg b)$ $R3 : G(\neg b \Rightarrow \neg u)$
 $A4 : G(u \Rightarrow F(a) \wedge u \Rightarrow c)$ $R4 : F(l \ U \ (u \ U \ c))$
 $F1 : F(a \wedge F(\neg a))$

Fig. 5. The two tables explain the shorthands for atomic propositions and list the LTL requirements

6 Conclusion

This paper further expands the incorporation of formal methods into software development. Aiming specifically at the requirements stage we propose a novel approach to sanity checking of requirements formalised in LTL formulae. Our approach is comprehensive in scope integrating consistency, vacuity and completeness checking to allow the user (or a requirements engineer) to produce a high quality set of requirements easier. The novelty of our consistency (vacuity) checking is that they produce all inconsistent (vacuous) sets instead of a yes/no answer and their efficiency is demonstrated in an experimental evaluation. Finally, the completeness checking presents a new behaviour-based coverage and suggests formulae that would improve the coverage and, consequently, the rigour of the final requirements.

One direction of future research is the pattern-based candidate selection mentioned above. Even though the selected candidates were relatively sensible in presented experiments, using random formulae can produce useless results. Finally, experimental evaluation on real-life requirements and subsequent incorporation into a toolchain facilitating model-based development are the long term goals of the presented research. This paper also lacks formal definition of *total coverage* (which the proposed partial coverage merely approximates). We intend to formulate an appropriate definition using uniform probability distribution: which would also allow to compute total coverage without approximation and would not be biased by concrete LTL to BA translation. That solution, however, is not very practical since the underlying automata translation is doubly exponential.

References

1. Barnat, J., Brim, L., Češka, M., Ročkal, P.: DiVinE: Parallel Distributed Model Checker. In: Proc. of HiBi/PDMC, pp. 4–7 (2010)
2. Blom, S., Fokkink, W.J., Groote, J.F., van Langevelde, I., Lisser, B., van de Pol, J.: μ CRL: A Toolset for Analysing Algebraic Specifications. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 250–254. Springer, Heidelberg (2001)
3. Chan, W., Anderson, R.J., Bea, P., Burns, S., Modugno, F., Notkin, D., Reese, J.D.: Model Checking Large Software Specifications. IEEE T. Software Eng. 24, 498–520 (1998)
4. Chockler, H., Kupferman, O., Kurshan, R.P., Vardi, M.Y.: A Practical Approach to Coverage in Model Checking. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 66–78. Springer, Heidelberg (2001)

5. Chockler, H., Kupferman, O., Vardi, M.Y.: Coverage Metrics for Temporal Logic Model Checking. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 528–542. Springer, Heidelberg (2001)
6. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
7. Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory-Efficient Algorithms for the Verification of Temporal Properties. *Form. Method Syst. Des.* 1, 275–288 (1992)
8. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property Specification Patterns for Finite-State Verification. In: Proc. of FMSP, pp. 7–15 (1998)
9. Heimdahl, M.P.E., Leveson, N.G.: Completeness and Consistency Analysis of State-Based Requirements. In: Proc. of ICSE, pp. 3–14 (1995)
10. Hinchey, M., Jackson, M., Cousot, P., Cook, B., Bowen, J.P., Margaria, T.: Software Engineering and Formal Methods. *Commun. ACM* 51, 54–59 (2008)
11. Industrial Framework for Embedded Systems Tools, <http://www.artemis-ifest.eu>
12. Kupferman, O.: Sanity Checks in Formal Verification. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 37–51. Springer, Heidelberg (2006)
13. Kupferman, O., Vardi, M.Y.: Vacuity Detection in Temporal Model Checking. *STTT* 4, 224–233 (2003)
14. Leveson, N.: Completeness in Formal Specification Language Design for Process-Control Systems. In: Proc. of FMSP, pp. 75–87 (2000)
15. Miller, S.P., Tribble, A.C., Heimdahl, M.P.E.: Proving the Shalls. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 75–93. Springer, Heidelberg (2003)
16. Rajan, A., Whalen, M.W., Heimdahl, M.P.E.: Model Validation using Automatically Generated Requirements-Based Tests. In: Proc. of HASE, pp. 95–104 (2007)
17. Roy, S., Das, S., Basu, P., Dasgupta, P., Chakrabarti, P.P.: SAT Based Solutions for Consistency Problems in Formal Property Specifications for Open Systems. In: Proc. of ICCAD, pp. 885–888 (2005)
18. Rozier, K.Y., Vardi, M.Y.: LTL Satisfiability Checking. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 149–167. Springer, Heidelberg (2007)
19. Tasiran, S., Keutzer, K.: Coverage Metrics for Functional Validation of Hardware Designs. *IEEE Des. Test. Comput.* 18(4), 36–45 (2001)
20. Whalen, M.W., Rajan, A., Heimdahl, M.P.E., Miller, S.P.: Coverage Metrics for Requirements-Based Testing. In: Proc. of ISSTA, pp. 25–36 (2006)

TVAL+ : TVLA and Value Analyses Together

Pietro Ferrara, Raphael Fuchs, and Uri Juhasz

ETH Zürich, Switzerland

{pietro.ferrara,uri.juhasz}@inf.ethz.ch, fuchsra@ethz.ch

Abstract. Effective static analyses must precisely approximate both heap structure and information about values. During the last decade, shape analysis has obtained great achievements in the field of heap abstraction. Similarly, numerical and other value abstractions have made tremendous progress, and they are effectively applied to the analysis of industrial software. In addition, several generic static analyzers have been introduced. These compositional analyzers combine many types of abstraction into the same analysis to prove various properties. The main contribution of this paper is the combination of *Sample*, an existing generic analyzer, with a TVLA-based heap abstraction (TVAL+).

1 Introduction

During the last decades, heap analysis has been extensively, deeply and successfully studied. Its goal is to approximate all possible heap shapes in a finite way. This is particularly important when analyzing object-oriented programs, which heavily interact with dynamically allocated memory. Static analysis has been widely applied to the abstraction of numerical information as well. Numerical domains [8,22] track static information at different levels of approximation. In addition, other approaches (e.g., string analyses) approximate other types of information over the values computed during the execution.

Usually, the combination of the heap abstraction with information about other values (called *value* domain) is necessary. For instance, consider a program that sum the values contained in the nodes of a list. Here we would like to prove that, at the end of the execution, the computed value of is the summation of all elements in the given list. For this reason, several recent approaches have combined heap and value abstractions. In this context, some heap analyses (e.g., TVLA) were extended with information about numerical values [21], or ad-hoc heap analyses were combined with some existing numerical domains [3].

Thanks to compositional analyses based on abstract interpretation [5], we can define a generic analyzer that combines various abstractions modularly and automatically. In this way, the implementation of different domains can be composed together without reimplementing the analysis. In addition, generic static analyzers take care of all aspects not strictly related to the abstract domain, e.g., the computation of a fixpoint. As far as we know, existing generic analyzers [10,9,24] apply a fixed heap analysis, while they let the user specify the value abstraction and the property of interest. *Sample* (Static Analyzer of Multiple Programming

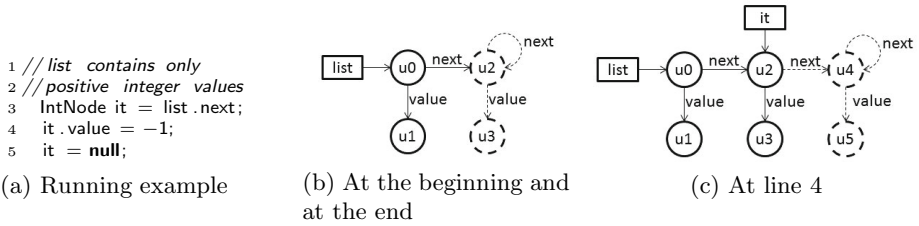


Fig. 1. The running example and the states obtained by TVLA when analyzing it

LanguagEs) is a novel generic analyzer of object-oriented programs that is parametric not only in the value domain and in the property of interest, but in the heap abstraction as well.

Contribution Given this context, the contribution of this work is the extension of *Sample* with a TVLA-based heap analysis (TVAL+). In particular, we formalize the structure of *Sample*, how we name nodes in TVLA states through name predicates, and how we communicate the modifications performed by TVLA on the heap structure to the value analysis. In this way, TVAL+ can be combined with any existing value abstraction in *Sample*. The combination between TVAL+ and value analyses is completely automatic.

Intuitively, *Sample* computes an abstract state for each program point. We use TVLA as the engine to define the heap small-step semantics of our language, while the value analysis tracks information over the values contained in abstract heap nodes represented by heap identifiers. Since TVLA names nodes in a completely unpredictable and arbitrary way, and it does not provide any information from where nodes come from after the application of a TVLA action, we augment standard TVLA states with name predicates, and we normalize the exit states obtained by applying TVLA actions to keep the naming schema consistent. When we perform this normalization, we communicate the changes performed by TVLA on the heap structure to the value analysis. Our approach supports this normalization without requiring any additional feature to the value analysis, since it relies on standard semantic operators (namely, assignment and forgetting of identifiers). The TVLA state can contain any instrumentation predicates.

1.1 Running Example

Consider now the code in Figure 1a. This program assigns -1 to the value contained in the second node of a given list. Let us suppose that the list is acyclic, it contains at least two nodes, and that the initial TVLA state is the one depicted in Figure 1b. The node pointed by `list.next` is materialized when it is assigned to the iterator, while `it.value` is materialized when it is assigned at line 4. Therefore, after the analysis of line 4, TVLA infers the heap state depicted in Figure 1c. Finally, when we assign null to `it`, TVLA removes this unary predicate

¹ TVLA would actually give more options, which we omit here.

| | | | |
|---|---|---|---|
| <pre> %s PVar {x,y,z,...} foreach (x in PVar) { %p x(v₁) unique } </pre> <p>(a) Program variable predicates</p> | <pre> %s Fields {n, i, ...} foreach (f in Fields) { %p f(v₁,v₂) function } </pre> <p>(b) Field predicates</p> | <pre> %action createObj(t) { %new { t(v) = isNew(v) } } </pre> <p>(c) Object creation</p> | <pre> %action getField(u,t,f) { %f { E(v₁,v₂) t(v₁) & f(v₁,v₂) } { u(v) = E(v₁) t(v₁) & f(v₁, v) } } </pre> <p>(d) Field access</p> |
| <pre> %action assignVariable(t,s) { %f { source(v) } { t(v) = s(v) } } </pre> <p>(e) Variable assignment</p> | <pre> %action assignField(t,f,s) { %f { t(v), s(v) } { f(v₁, v₂) = (f(v₁, v₂) & !t(v₁)) (t(v₁) & s(v₂)) } } </pre> <p>(f) Field assignment</p> | <pre> %action lub() { { } } </pre> <p>(g) Upper bound</p> | |

Fig. 2. TVLA actions of the heap semantics

from the TVLA state, and this leads to summarize u_2 with u_4 , and u_3 with u_5 . This brings the analysis to the initial state depicted in Figure 15.

2 Background

2.1 Sample

Sample (Static Analyzer of Multiple Programming Languages) is a novel generic analyzer of object-oriented programs based on the abstract interpretation theory [6,7]. Relying on compositional analyses, Sample can be instantiated with various heap abstractions and value domains. A state of the analysis is a pair composed by a state of the heap domain H and a state of the value domain V (formally, $\Sigma = H \times V$). Sample has been already applied to various value analyses [4,11,12,25], and it supports some of the most common numerical analyses through Apron [17]. In addition, some rough heap analyses have been already developed in Sample. The analyzer works on an intermediate object-oriented language called Simple, and it supports the compilation of Scala and Java bytecode to this language. Simple is based on control flow graphs (cfg). Each block of the cfg contains a list of statements that may be $x := y.f$, $y.f := x$, or $x := \text{new } T$.

2.2 Shape Analysis

TVLA [18] is a framework for defining and implementing heap abstractions in 3-valued first order logic [23] with transitive closure (FOLTC). In this section, we sketch the standard TVLA features we adopt in TVAL+. For each analysis a FOL signature (predicates of arity up to 2) is defined. The predicates are divided into core (uninterpreted) predicates, and instrumentation predicates, which are defined by a FOLTC formula over the core predicates. The abstract domain is composed of sets of structures of 3-valued FOLTC. A 3-valued structure is composed of normal and *summary* nodes. Normal nodes represent exactly one concrete node, while summary nodes may represent many concrete nodes.

The abstraction is defined by a set of unary predicates out of the signature, which can be core or instrumentation predicates (the *abstraction predicates AP*). In a normalized structure any two distinct nodes are differentiated by at least one abstraction predicate. For heaps, usually unary predicates represent local reference valued variables and binary predicates represent reference valued fields. The graphical representation, as in Figure 11b, uses circles for nodes (dashed for summary nodes), labeled originless arrows for local variables (again dashed for may point to), and labeled arrows between nodes to represent reference valued fields. For example, for a linked list as in Figure 11b, we could use as abstraction predicates with one free variable pointed-to-by-list (u0 in the example) $list(x)$, so that the first node is not summarized with the rest of the nodes.

In our example, we would need predicates to differentiate u1, u3 and u5. We can achieve that by using (i) value-of-node-pointed-to-by-list (u1), that is, $\exists y : list(y) \wedge value(y, x)$, (ii) value-of-node-pointed-to-by-it (u3 in 11c), that is, $\exists y : it(y) \wedge value(y, x)$, and (iii) $\exists y, z : it(y) \wedge next * (y, z) \wedge value(z, x)$ for differentiating the nodes coming before and after it.

Concrete transformers are represented by update formulae ($P'(\bar{x}) = \phi(\bar{x})$). Here P is a predicate symbol, ϕ a formula (evaluated in the pre-state), \bar{x} are bound variables (implicitly universally quantified - exactly as many as the arity of P) and P' is P in the post state. For example, Figure 21 represents the assignment $t.f = s$. Here the new value of f (the field being written) is given by a formula on the old values of f, t and s (we omit here null checks).

TVLA works by applying a *semantic reduction* (called *focus*) before the abstract transformer. Given an abstract state, focus produces a set of abstract states with the same concrete representation, but ensuring some pairs of nodes are not merged, in addition to the separation enforced by the abstraction predicates. In the linked list example, before advancing to the next node, we would like to make sure it is not merged with any other node, so we would focus on it using the formula $\exists y : it(y) \wedge next(y, x)$. TVLA uses *widening* (called *blur*) to ensure termination. After applying the abstract transformer on the focused structures, nodes which are not separable by the abstraction predicates are merged, and the same happens for structures, ensuring a bound on the size of the abstract domain.

TVLA Actions

Figure 2 reports all TVLA actions that are used in TVAL+. For every program variable x , a unary predicate $P_x(v)$ specifies the node that is pointed by the local variable (Figure 2a). Unique specifies that at most one node satisfies the predicate. Heap nodes are connected to each other when a field of an object references another object. For every possible field f , we introduce a binary predicate $P_f(v_1, v_2)$ that connects heap nodes (Figure 2b). For example, if field n of node a references node b , we have that $P_n(a, b) = 1$. In the definition, function means that the field relation is a (partial) function. Object creation relies on the TVLA built-in predicate *isNew*. It creates a new (non-summary) node and assigns it to a temporary program variable **temp** (Figure 2c). When we access a field of an object, the node modelling the target object may have been summarized with

other nodes. However, we would like to obtain a concrete (i.e., not summarized) node as the result of our access. With this purpose we add the focus formula $\exists(v_1, v_2) : P_{target}(v_1) \wedge P_f(v_1, v_2)$. As in the case of object creation, the result is assigned to a temporary program variable (Figure 2d). In the case of assignments, we assume there is always a unary predicate pointing to the source of the assignment. In the case of a variable, it is the program variable predicate, while in the case of a heap access or an object creation it is the variable `temp` which was created as explained above. Therefore, we simply copy the valuation of the unary predicate (Figure 2e). The treatment of the assignment to a field is similar to normal assignment. However, the translation to TVLA is different, as it involves a field predicate, and we need to access the target object whose field is assigned (Figure 2f). When we join two states (e.g., when computing the exit state of an if statement), we rely on the join performed automatically by TVLA on all input structures in the entry state. Therefore, we simply provide TVLA with the two states and an empty action, and we take the exit state as the result of the upper bound operator (Figure 2g).

3 Heap and Value Analyses in Sample

The state of the computation of an object oriented program can be defined as the combination of the heap structure with the values that can be contained in heap locations or local variables. Let `Ref` be the set of concrete references, and `FieldName` the set of field names. The heap may be defined by `Ref` \times `FieldName` \rightarrow `Ref` for heap locations, and by `Varld` \rightarrow `Ref` (where `Varld` is the set of local variables) for the local variables. Let `Val` be the set of values (e.g., integers or strings). The runtime values can be represented by `Ref` \times `FieldName` \rightarrow `Val` for heap locations, and by `Varld` \rightarrow `Val` for local variables.

When we reason about the abstraction of concrete states, often we would like to reason about heap structures and values separately. Therefore, we suppose that concrete references are abstracted by abstract heap identifiers (`HId`). Each heap analysis defines its own finite set of heap identifiers. A heap identifier could represent one or many concrete references. Let $\gamma_{\text{HId}} : \text{HId} \rightarrow \wp(\text{Ref})$ be the concretization of abstract heap identifiers. We say that a heap identifier `i` represents a summary node if $|\gamma_{\text{HId}}(i)| > 1$.

Since the heap analysis needs to abstract together many concrete heaps, a heap interaction (e.g., a field access) may provide many heap identifiers. Sometimes the heap analysis may not be able to establish one exact node for a heap access, and therefore it would return a *possible* set of heap identifiers. In other cases, the heap analysis could track disjunctive information through a set of heap states at a given program point, and therefore it would return a *definite* set of heap identifiers. Formally, we define set of heap identifiers `SHId` = $\wp(\text{HId}) \times \{\text{true}, \text{false}\}$. The boolean flag is `true` if the set is definite (if it represents *all* identifiers in the set), `false` if it is possible (if it represents *some* of them). Note that trace partitioning [20] is supported in Sample [13]. Therefore, if the heap domain uses a disjunction of heaps rather than a single 3-valued logical structure (as often happens in TVLA), we can use this feature to prove complex properties.

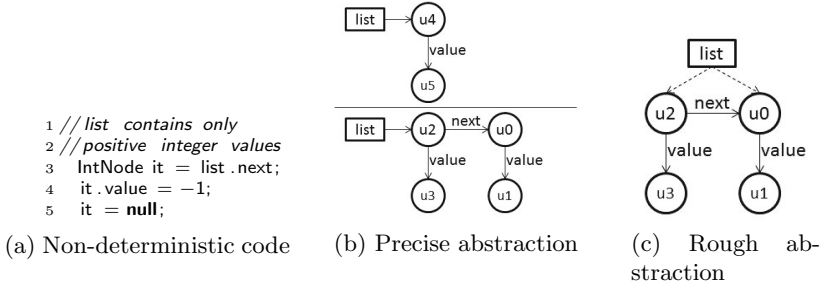


Fig. 3. A non-deterministic program possible abstractions of the heap at line 7

Example: Consider the code in Figure 3a. This program non-deterministically adds a node at the beginning of a list containing only one element. It then assigns 1 to field `value` of the node at the head of the list. Heap analyses could produce different states at line 7. A precise analysis like TVLA may track two distinct states (Figure 3b), while a rough analysis may abstract the two states into one (Figure 3c). What happens when we assign to `list.value`? In the first scenario, we have to perform a strong assignment to both the nodes pointed by `list.value`. This is represented by assigning to the definite set of heap identifiers ($\{\{u5, u3\}, \text{true}\}$). In the second scenario, we can only perform a weak assignment, since we do not have two distinct states. This is represented by the possible set ($\{\{u1, u3\}, \text{false}\}$).

3.1 Replacements

The application of semantic or lattice operators could affect the structure and the identifiers contained in the heap state. In particular, nodes could be materialized or merged. We must reflect these changes in the state of the value domain to preserve the soundness of the analysis. This information is passed by functions in $R = \wp(\text{HId}) \rightarrow \wp(\text{HId})$ called *replacements*. Given a single relation in a replacement, its semantics is to assign the upper bound of the values of identifiers on the left side to all identifiers in the right side.

Running Example: Consider the transition from Figure 1b to Figure 1c. The node `u2` of the initial state is split into nodes `u2` and `u4` in the final state. This is represented by the relation $\{u2\} \mapsto \{u2, u4\}$. The same happens on `u3` when it is split to nodes `u3` and `u5`. Therefore we obtain the replacement $[\{u2\} \mapsto \{u2, u4\}, \{u3\} \mapsto \{u3, u5\}]$. Consider now the transition obtained analyzing `it = null`, that is, the transition from Figure 1c to Figure 1d. `u2` and `u4` are summarized into `u2`. The same happens for `u3` and `u5` that are summarized to `u3`, obtaining the replacement $[\{u2, u4\} \mapsto \{u2\}, \{u3, u5\} \mapsto \{u3\}]$. \square

3.2 Heap Analysis

The semantic operators of the heap analysis are (i) $getFieldId_H : (\text{VarId} \times \text{FieldName} \times H) \rightarrow (\text{SHId} \times H \times R)$, (ii) $assignVar_H : (\text{VarId} \times \text{SHId} \times H) \rightarrow$

$$\begin{aligned}
& \text{assignVar}_V : (\text{VarId} \times \text{SHId} \times V) \rightarrow V \\
& \text{assignVar}_V(x, (l, b), s) = \begin{cases} s & \text{if } l = \emptyset \\ \bigsqcup_{i \in l} \text{assignId}_V(x, i, s) & \text{if } b = \text{false} \\ \bigsqcap_{i \in l} \text{assignId}_V(x, i, s) & \text{if } b = \text{true} \end{cases} \\
& \text{assignHIds}_V : (\text{SHId} \times \text{VarId} \times V) \rightarrow V \\
& \text{assignHIds}_V((l, b), x, s) = \begin{cases} s & \text{if } l = \emptyset \\ \bigsqcup_{i \in l} \text{assignId}_V(i, x, s) & \text{if } b = \text{false} \\ \bigsqcap_{i \in l} \text{assignId}_V(i, x, s) & \text{if } b = \text{true} \end{cases} \\
& \text{replace}_V : (V \times R) \rightarrow V \\
& \text{replace}_V(s, \emptyset) = s \\
& \text{replace}_V(s, r) = \text{forgetAll}_V((\bigsqcup_{l \in \text{dom}(r)} l) \setminus (\bigsqcup_{l \in \text{dom}(r)} r(l)), s_n) \\
& \quad \text{where } \text{dom}(r) = \{l_1, \dots, l_n\} \wedge s_0 = s \wedge \forall i \in [1..n] : \\
& \quad \quad s'_i = \text{assignVar}_V(\text{temp}, (l_i, \text{false}), s_{i-1}) \wedge \\
& \quad \quad s''_i = \text{assignHIds}_V(r(l_i), \text{true}, \text{temp}, s'_i) \wedge \\
& \quad \quad s_i = \text{forget}_V(\text{temp}, s''_i) \\
& \text{and } \text{forgetAll} : (\emptyset(\text{Id}) \times V) \rightarrow V \text{ is defined as follows:} \\
& \text{forgetAll}_V(\{i_1, \dots, i_n\}, s) = s_n \text{ where } s_0 = s \wedge \forall k \in [1..n] : \\
& \quad s_k = \text{forget}_V(i_k, s_{k-1})
\end{aligned}$$

Fig. 4. Definition of replace_V

($H \times R$), (iii) $\text{assignField}_H : (\text{VarId} \times \text{FieldName} \times \text{VarId} \times H) \rightarrow (H \times R)$, and (iv) $\text{createObject}_H : (C \times H) \rightarrow (\text{SHId} \times H \times R)$. C is the set of classes that can be instantiated. All these operators return a state of the heap, and a replacement to represent merges and materializations of heap nodes. In addition, $\text{getFieldId}_H(x, f, h)$ returns a set of heap identifiers that could be pointed to by $x.f$ in h . $\text{assignVar}_H(x, l, h)$ assigns l to x , while $\text{assignField}_H(x, f, l, h)$ assigns l to $x.f$. Finally, $\text{createObject}_H(C, h)$ creates an instance of class C returning the heap identifiers pointing to the fresh object as well.

3.3 Value Analysis

The value analysis treats variable and heap identifiers in the same way. Therefore we define identifiers (Id) as variable (VarId) or heap (HId) identifiers ($\text{Id} = \text{VarId} \cup \text{HId}$). The semantic operators the value analysis has to provide are (i) $\text{assignId}_V : (\text{Id} \times \text{Id} \times V) \rightarrow V$, and (ii) $\text{forget}_V : (\text{Id} \times V) \rightarrow V$. $\text{assignId}_V(x, y, v)$ assigns the value of y to x in state v , while $\text{forget}_V(x, v)$ removes the value of x from state v . Usually these operators are already supported by existing value analyses. The only additional feature the value analysis has to take into account is when a single heap identifier represents a summary node performing weak updates.

Relying on these semantic operators, Figure 4 defines how replacements and assignments are computed by **Sample** in the value analysis. These operators will be used in the definition of the semantics of our language. $\text{assignVar}_V(x, l, s)$ assigns the set of heap identifiers contained in l to variable x , while $\text{assignHIds}_V(l, x, s)$ assigns variable x to the set of heap identifiers inside l . Both these functions behave in accordance with whether the set of heap identifiers is definite or possible. If we assign a definite set of heap identifiers to a variable, we assign the *greatest* lower bound of the values of all given heap identifiers (since we have to assign the intersection of their values). On the other hand, if we assign a possible set, we assign *one* of the values, and therefore we have to

| | | |
|---|---|--|
| <ol style="list-style-type: none"> 1 $\mathbb{S}[\![x := y.f, (h, s)]\!] = (h_2, s_3) :$ 2 $getFieldId_H(y, f, h) = (l, h_1, r) \wedge$ 3 $assignVar_H(x, l, h_1) = (h_2, r_1) \wedge$ 4 $replace_V(s, r) = s_1 \wedge$ 5 $replace_V(s_1, r_1) = s_2 \wedge$ 6 $assignVar_V(x, l, s_2) = s_3$ | <ol style="list-style-type: none"> 1 $\mathbb{S}[\![y.f := x, (h, s)]\!] = (h_2, s_3) :$ 2 $assignField_H(y, f, x, h) = (h_1, r) \wedge$ 3 $getFieldId_H(y, f, h_1) = (l, h_2, r_1) \wedge$ 4 $replace_V(s, r) = s_1 \wedge$ 5 $replace_V(s_1, r_1) = s_2 \wedge$ 6 $assignHIds_V(l, x, s_2) = s_3$ | <ol style="list-style-type: none"> 1 $\mathbb{S}[\![x := new T, (h, s)]\!] = (h_2, s_3) :$ 2 $createObject_H(T, h) = (l, h_1, r) \wedge$ 3 $assignVar_H(x, l, h_1) = (h_2, r_1) \wedge$ 4 $replace_V(s, r) = s_1 \wedge$ 5 $replace_V(s_1, r_1) = s_2 \wedge$ 6 $assignVar_V(x, l, s_2) = s_3$ |
|---|---|--|

Fig. 5. Sample’s semantics of statements

take the upper bound. Similarly, when we assign a variable to a possible set of heap identifiers, we are affecting only one of the heap identifiers in the set, and therefore we have to take the upper bound. Instead, when we are assigning to a definite set, we take the greatest lower bound of the assignments of all the heap identifiers.

$replace_V(s, r)$ applies the replacement r to s . For each relation $[D \mapsto C] \in r$ it assigns the upper bound of the values of identifiers in D to each identifier in C . In its definition we denote by $temp \in \mathit{VarId}$ a variable identifier that does not appear in the program. This variable is used as a gateway to build up the abstract value represented by the variables in D , and to assign it to all identifiers in C . At the end we remove all identifiers that appear at least once on the left part of the replacement, and never on the right side. Intuitively, these identifiers are replaced by something else, and they are never used as target of other replaced variables. Therefore, they are not anymore used, and they can be safely removed.

Running Example: We suppose that all nodes of the given list contain values greater or equal to zero at the beginning of the program in Figure 1a. Assuming we analyze the program using intervals (that is, tracking the interval of numerical values that each variable could have at a given program point), in the heap state of Figure 1b the value domain tracks that $[u1 \mapsto [0..\infty], u3 \mapsto [0..\infty]]$. After the first statement, the replacement we have to apply contains the relation $\{u3\} \mapsto \{u3, u5\}$. Therefore, the application of this replacement results in the state $[u1 \mapsto [0..\infty], u3 \mapsto [0..\infty], u5 \mapsto [0..\infty]]$. The semantics of $it.value = -1$ assigns $[-1.. -1]$ to $u3$ obtaining the state $[u1 \mapsto [0..\infty], u3 \mapsto [-1.. -1], u5 \mapsto [0..\infty]]$. When we finally apply the second replacement during the evaluation of $it = null$, the relation $\{u3, u5\} \mapsto \{u3\}$ tells the analysis to (i) assign the upper bound of $u3$ and $u5$ (that is, $[-1..\infty]$) to $u3$, and (ii) remove $u5$. Therefore, the final state is $[u1 \mapsto [0..\infty], u3 \mapsto [-1..\infty]]$. \square

3.4 Overall Semantics

Figure 5 defines the semantics of the language introduced in Section 2.1. When we assign $y.f$ to x , we extract the identifiers pointed by $y.f$ (line 2) and we assign them to x in the state of the heap analysis (line 3). These two actions lead to two distinct replacements, which are passed to the value domain (line 4 and 5) before assigning the identifiers of $y.f$ to x (line 6). Similarly, when we assign x to $y.f$, we perform the assignment on the heap state (line 2), and we query the heap analysis to obtain the identifiers pointed by $y.f$ (line 3). The two actions produce two replacements that are passed to the value domain (line 4 and 5). At the end, x is assigned to the heap identifiers pointed by $y.f$ in the value domain (line 6).

When we assign new \top to x , we create the object (line 2) and we assign it to x in the heap analysis (line 3) obtaining two replacements. After the application of these two replacements in the value domain (line 4 and 5), the heap identifiers of the created object is assigned to x in the value domain (line 6).

4 TVAL+

We need that each node is represented exactly by one heap identifier in a TVLA state, and each heap identifier points exactly to one node. Since TVLA names nodes in a unpredictable way, and the same node could have several canonical (that is, the evaluation of abstract predicates) names, we need to add some predicates in order to track node identity. In addition, TVLA does not provide any information about from where nodes come after an action, while these predicates track that. Note that so far we did not use the names given by TVLA in the running example, but we adopted a more predictable naming schema.

4.1 Name Predicates

We name nodes through unary non-abstraction predicates (called *name predicates*). Each time we run TVLA, the entry state contains a name predicate for each node. After running TVLA, name predicates tell us how nodes have been split or merged. We then *normalizes* the exit state to ensure that each node is pointed to exactly by one name predicate, and each name predicate points exactly to one node.

Usually unary predicates are used to distinguish between different structures when a join of heap states is performed. Since we only wish to track nodes, we do not want the name predicates to influence the abstraction. We achieve this behavior through non-abstraction predicates, since these allow nodes to be merged even though different non-abstraction predicates hold for them [18].

The naming schema defines how we name nodes. A naïve approach is to consecutively number all created heap nodes. The numbers are based on the pre-state, and not counted globally, since otherwise we could go on creating new names endlessly. In addition, we always need to obtain the same result when the same operation is performed on the same pre-state. If we used a global counter, this property would not be guaranteed. However, we would lose a lot of precision in the analysis with this approach. Consider for instance an if statement that allocates an object in both branches, but it assigns -1 to its `value` field in a branch, and 1 in the other. Suppose that the internal counter of the state of the analysis is 0. The analysis would name the nodes created by the new statements in the two branches with the same name (that is, 1 for the created object, and then 2 for its field `value`). When the abstract states in the two branches are joined to compute the abstract state after the if statements, the values associated with the heap identifier 2 in the value domain are joined as well. This means that, regardless of the fact that the two nodes could be kept disjoint by the heap analysis, we merged the values of the two nodes in the value domain,

introducing a sensible loss of precision. In the example above, we would not be able to distinguish that value 1 may have been assigned only to `x.value`, and -1 only to `y.value` after the `if` statement.

4.2 TVAL+ Naming Schema

This example shows that we need a more sophisticated naming schema. In particular, we have to take into account the context in which heap identifiers are created. Therefore, the name of a new node is based on the allocation site. In addition, since a given program point pp could create several nodes (e.g., inside a `while` loop), we have to count the number of times we are allocating (in the abstract) the node, and increment the counter at each iteration. Let PP be the set of program points. A *basic* heap identifier is a pair composed of a program point and a natural number (formally, $\text{BHid}_{\text{TVAL}+} = \text{PP} \times \mathbb{N}$).

What happens when two nodes are merged into a summary node? Since TVLA could summarize nodes created at different program points, we have to extend the definitions above to precisely approximate this scenario. Therefore, a heap identifier is composed by a set of basic heap identifiers. When TVLA summarizes two nodes created at different program points (e.g., $(\text{pp1}, n_1)$ and $(\text{pp2}, n_2)$), the resulting name will be a set composed by both $(\{(\text{pp1}, n_1), (\text{pp2}, n_2)\})$.

Instead, when a node is materialized from a summary node, the output state of TVLA will contain two nodes pointed by the same name predicate, and we have to provide two different names when normalizing this state. To keep the precision of the analysis in this scenario, we add another counter to the whole heap identifier. Formally, the set of heap identifiers is defined by $\text{Hid}_{\text{TVAL}+} = \wp(\text{BHid}_{\text{TVAL}+}) \times \mathbb{N}$. **Normalization.** By normalized state we mean a state in which (i) each node is pointed only by one name predicate, and (ii) each name predicate points only to one node. After we run TVLA on a normalized state, we may obtain a state in which a name predicate points to many nodes, and a node is pointed by many name predicates. Figure 6 formalizes this normalization. We focus the formal definitions on the part of the TVLA state that deal with name predicates. Let $\text{Nodes}_{\text{TVAL}+}$ be the set of node identifiers given by TVLA. We define the TVAL+ state by a function that relates each node to the set of name predicates that point to it. Formally, $\Sigma_{\text{TVAL}+} = \text{Nodes}_{\text{TVAL}+} \rightarrow \wp(\text{Hid}_{\text{TVAL}+})$. Function π_{Id} returns the set of all heap identifiers contained in a given state, while rev returns a function relating each heap identifier to the set of nodes it may point to.

First of all, we define what it means to merge a set of name predicates when they point to the same node. *mergeHids* takes the set union of all program points in the set of heap identifiers, and the minimum values for the counters (point *b*). The same approach is adopted for the global counter of the heap identifier (point *a*). *mergeHids* is the basis to define the merge of a complete state performed by *merge*. This function applies *mergeHids* to all nodes that are pointed by more than one name predicate, and it builds up a coherent replacement function (point *c*). After that, we have that each node is pointed to by one name predicate, but we do not have yet that each name predicate points only to a node. *split* ensures this. For each predicate name that points to (at least) two nodes, and for each

$$\begin{aligned}
\pi_{\text{id}} &: \Sigma_{\text{TVAL}^+} \rightarrow \wp(\text{Hld}_{\text{TVAL}^+}) \\
\pi_{\text{id}}(f) &= \bigcup_{n \in \text{dom}(f)} f(n) \\
\\
\text{rev} &: \Sigma_{\text{TVAL}^+} \rightarrow (\text{Hld}_{\text{TVAL}^+} \rightarrow \wp(\text{Nodes}_{\text{TVAL}^+})) \\
\text{rev}(f) &= [\text{id} \mapsto P : \text{id} \in \pi_{\text{id}}(f) \wedge P = \{p' : \text{id} \in f(p')\}] \\
\\
\text{mergeHlds} &: \wp(\text{Hld}_{\text{TVAL}^+}) \rightarrow \text{Hld}_{\text{TVAL}^+} \\
\text{mergeHlds}(\{E_i, u_i : i \in [0..n]\}) &= (E', \min(\{u_i : i \in [0..n]\})) : & (a) \\
E' = \{(\text{pp}, c) : \exists (\text{pp}, c') \in \bigcup_{i \in [0..n]} E_i \wedge c = \min(\{c' : (\text{pp}, c') \in \bigcup_{i \in [0..n]} E_i\})\} & & (b) \\
\\
\text{merge} &: \Sigma_{\text{TVAL}^+} \rightarrow (\Sigma_{\text{TVAL}^+} \times \mathbb{R}) \\
\text{merge}(f) &= (f', r) : \\
f' &= \left[n \mapsto \begin{cases} \{\text{mergeHlds}(f(n))\} & \text{if } |f(n)| > 1 \\ f(n) & \text{if } |f(n)| = 1 \end{cases} : n \in \text{dom}(f) \right] \wedge \\
r &= [f(n) \mapsto \{\text{mergeHlds}(f(n))\} : \exists n \in \text{dom}(f) : |f(n)| > 1] & (c) \\
\\
\text{split} &: \Sigma_{\text{TVAL}^+} \rightarrow (\Sigma_{\text{TVAL}^+} \times \mathbb{R}) \\
\text{split}(f) &= (f', r) : \\
f' &= \left[\begin{array}{l} n \mapsto P : n \in \text{dom}(f) \wedge \\ P = \begin{cases} \{(T, i) \mid \{k \in \text{exclude}((T, i), f) : k \leq i\} + \text{in}(n, N))\} & \text{if } f(n) = \{(T, i)\} \wedge \\ & \text{rev}(f)(T, i) = N \wedge |N| > 1 \\ f(n) & \text{otherwise} \end{cases} \end{array} \right] & (d) \\
r &= [\{(T, i)\} \mapsto \{(T, i') : \exists n : f(n) = \{(T, i)\} \wedge \text{rev}(f)(T, i) = N \wedge |N| > 1 \\ & \quad i' \in \bigcup_{n' \in N} \{\text{exclude}((T, i), f) : k \leq i\} + \text{in}(n', N)\} : |\text{rev}(f)(T, i)| > 1] \\
\\
\text{in} &: \text{Nodes}_{\text{TVAL}^+} \times \wp(\text{Nodes}_{\text{TVAL}^+}) \rightarrow \mathbb{N} \\
\text{in}(n, N) &= |\{n' \in N : n' <_{\text{TVAL}^+} n\}| \\
\\
\text{exclude} &: \text{Hld}_{\text{TVAL}^+} \times \Sigma_{\text{TVAL}^+} \rightarrow \wp(\mathbb{N}) \\
\text{exclude}((T, i), f) &= \{j : j \neq i \wedge |\text{rev}(f)((T, i))| = 1\} \\
\\
\text{normalize} &: \Sigma_{\text{TVAL}^+} \rightarrow (\Sigma_{\text{TVAL}^+} \times \mathbb{R}) \\
\text{normalize}(f) &= (f', r) : (f_1, r_1) = \text{merge}(f) \wedge (f', r_2) = \text{merge}(f_1) \wedge r = \text{combine}(r_1, r_2)
\end{aligned}$$

Fig. 6. Formal definition of the normalization of a TVLA state, where *combine*, given two replacements, builds up a replacement that is their concatenation

pointed node by this predicate, it creates a unique predicate by modifying its counter (point *d*), and it builds up a coherent replacement. The modification of the counter relies on *in*, a function that, given a set of nodes, and a node in that set, returns its position inside that set. To achieve this, we suppose that a total order $<_{\text{TVAL}^+}$ over node identifiers is provided by TVLA. In addition, *exclude* provides the set of the counters already used in other name predicates that point only to one node, and therefore they will be in the normalized state. In this way, we cover possible holes in the counters related to a given set of basic heap identifiers, avoiding duplicates and ensuring that the set of heap identifiers is bounded. Finally, *normalize* returns the normalized form of a given state by applying *merge* and *split* in sequence, and combining the two replacements returned by these two functions.

Running Example: When we analyze the running example introduced in Section 4.1, we start from the TVLA state represented in Figure 7a. The name predicates contain four basic heap identifiers $\{p1, p2, p3, p4\}$ to name the nodes in the entry state. Their initial counter is always zero. After the analysis of line 4, we obtain the state depicted in Figure 7b. Here nodes *u4* and *u5* have been materialized from nodes *u2* and *u3* respectively, and the name predicates

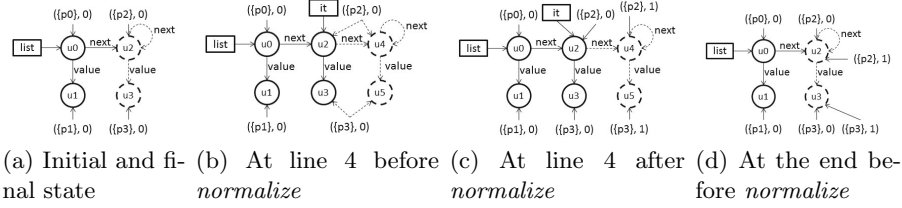


Fig. 7. TVLA states of the running example with name predicates

$(\{p2\}, 0)$ and $(\{p3\}, 0)$ both point to many (two) nodes. Therefore *normalize* introduces $(\{p2\}, 1)$ and $(\{p3\}, 1)$, and it sets them to point to one of the two nodes (Figure 7c). This is the entry state of the semantics of $it = \text{null}$. After the computation of the semantics of this statement, we obtain the state in Figure 7d. Here we have that $(\{p2\}, 0)$ and $(\{p2\}, 1)$ refer to the same node $u2$ (and the same happens with $(\{p3\}, 0)$ and $(\{p3\}, 1)$). Therefore, *normalize* merges these name predicates together into $(\{p2\}, 0)$, obtaining the same state we had at the beginning of the method (Figure 7a). Note that the heap identifier $(\{p2\}, 0)$ in the entry state represents something different in the exit state, since here it is the result of the merge of $\{(\{p2\}, 0), (\{p2\}, 1)\}$ as expressed by the replacement $\{(\{p2\}, 0), (\{p2\}, 1)\} \mapsto \{(\{p2\}, 0)\}$. \square

4.3 Abstract Semantics

The abstract semantics applies the TVLA actions introduced in Section 2.2 and normalizes the resulting states to define the semantic operators introduced in Figure 3.2. In particular, (i) $getFieldId_H$ applies the TVLA action in Figure 2d, (ii) $assignVar_H$ that in Figure 2e, (iii) $assignField_H$ that in Figure 2f, and (iv) $createObject_H$ that in Figure 2c. In all the cases, after the application of the TVLA action, the state is normalized through *normalize*. The same happens for the lattice operators. In addition, $createObject_H$ and $getFieldId_H$ return the heap identifier of the node pointed by *temp*.

5 Experimental Results

We ran TVAL+ on a set of case studies that represent a comprehensive set of common interactions with the heap and some representative examples of list manipulation. We combined TVAL+ with an implementation of intervals as value domain. We ran the analysis on an Intel Core 2 Duo CPU at 2.53GHz with 4GB of RAM. We used the Java HotSpot 64-Bit Server VM included in Java SE Runtime Environment 1.6.0_26-b03. Table 1 depicts the experimental results. Column $\#tvla$ shows the number of invocations of TVLA performed during the analysis, while **t** reports the time of execution of the analysis.

The analysis is quite fast in many cases, since the execution rarely requires more than few seconds. Anyway, most of these case studies are composed of few

Table 1. Execution times

| Program | #tvla | t (sec) | Program | #tvla | t (sec) |
|------------------------------|-------|---------|--------------------------------|-------|---------|
| createObject | 4 | 0.7 | createObjectIfCondition | 7 | 0.2 |
| accessNullField | 7 | 0.3 | assignFieldSelf | 7 | 0.3 |
| assignNumericField | 8 | 1.3 | assignNextField | 9 | 0.6 |
| createAndOverWrite | 9 | 0.7 | assumeEqual | 10 | 0.2 |
| assumeUnequal2 | 10 | 0.2 | overwriteField | 10 | 0.4 |
| assignAndAccessNumericField | 10 | 1.0 | conditionalAssignment | 11 | 0.4 |
| assignTwoFields | 12 | 0.9 | assumeUnequal | 13 | 0.3 |
| conditionalAssignmentVariant | 14 | 0.6 | appendByFieldAccessTwo | 15 | 0.6 |
| createSharedObject | 19 | 0.8 | buildList | 20 | 0.5 |
| appendByFieldAccessThree | 22 | 0.9 | createOneOrTwoNodes | 22 | 1.8 |
| accessNextSummarized | 23 | 0.6 | swapHeapObjectsOnce | 24 | 1.8 |
| createThreeElementList | 30 | 2.0 | linkObjects | 32 | 1.4 |
| createSummarizedIntList | 32 | 2.0 | swapLoop | 34 | 0.9 |
| linkAndTraverseObjects | 38 | 1.9 | createObjectWhile | 39 | 1.1 |
| assignFieldsAndSummarize | 44 | 4.1 | createPrependList | 54 | 1.4 |
| assignAndAddFields | 55 | 8.3 | appendByFieldAccessFour | 72 | 4.7 |
| traverseFixedShortList | 93 | 3.3 | createAppendList | 103 | 3.3 |
| initializeFixedList | 109 | 4.5 | createNumericalList | 148 | 14.4 |
| traverseSummarizedList | 164 | 8.6 | initializeAbstractedListFields | 220 | 31.8 |
| sumListElementsZero | 609 | 86.7 | | | |

sequential statements, and the examples whose analysis is slower are the ones that create a list and iterate over it. This means that the analysis has to compute a fixpoint, and this explains why TVLA is invoked many times.

For each of the case studies, we checked by hand if the heap abstraction produced by TVAL+ is what we expected, and if the information tracked by the value domain was sound and precise. In all examples, we obtained the expected results. For instance, in program `sumListElementsZero` we are able to (i) construct and precisely summarize a list whose nodes all contain 0 in field `value`, (ii) traverse the list computing the sum of the values, and (iii) prove that the sum of the elements is zero at the end of the program. This underlines that the combination of TVAL+ and the intervals domain fully benefits of the precision of TVLA, leading to really precise results on the value analysis as well.

6 Related Work

In this paper, we used an existing, well-established shape analysis to improve the results of other value analyses supported by `Sample`. McCloskey et al. [21] introduced a general way of integrating various analyses represented in FOLTC, combining different theories in a generic way. Their work allows the flow of information between all analyses concerned. To allow this flow, each analysis has to define classification and communication predicates, which are defined in terms of other predicates as well as core predicates that are interpreted only in that particular domain. Therefore, all analyses have to be represented in FOLTC. In our work, we took a different approach. On the one hand, we propagate the information only *from* the heap *to* the value analysis. On the other hand, we completely automate the integration between the two analyses without enforcing any restrictions on the value domain, which could track information that is not represented in FOLTC, thus allowing easier use of existing analyses. In addition, since the information flows only in one direction, this leads to faster analyses, as we only need to propagate information once.

Gopan et al. [15] presented a framework to track numeric information on array elements. This work is specific for numeric analyses over arrays, while our

approach targets any value analysis. A previous work [14] shows that its instantiation to a specific numeric domain is neither trivial nor automatic. The approach they adopted to reflect the modifications performed by the heap analysis is similar to ours. While they discharge the folding and unfolding of identifiers on the interface of the numerical analysis by adding some specific operators (namely, fold, expand, add, drop), our approach relies on assignment and forget.

Gulwani and Tiwari [16] combined analyzers represented in first order logic through an approach based on the Nelson-Oppen method. They propagate equalities both ways, but they place some restrictions on theories (e.g., convexity).

Magill et al. [19] adopted a shape analysis based on separation logic. Numerical domains are used to refine the heap analysis through counter-examples generated by the shape analysis. A potential error discovered by the shape analysis is translated into a counter-example program which is later reduced to a heapless arithmetic program. This program passes through the arithmetic analyzer in order to try and rule out the error by finding some arithmetic properties. This means they use arithmetic information only on demand to help to resolve potential heap errors, and not to prove arithmetic properties in general.

Beyer et al. [2] combined the model checker BLAST [1] with TVLA using Counter-Example Guided Abstraction Refinement for refining the shape analysis. Instead, we allow general abstract domains (not just predicate abstraction as in BLAST) at the price of having a fixed heap abstraction for the entire session.

Bouajjani et al. [3] developed a framework to statically infer properties over programs manipulating lists containing integer numerical data. This approach combines a specific heap analysis tracking information over lists with some existing numerical domains. Therefore, it cannot be automatically applied to other value analyses, or to analyze other heap structures, but it can prove properties that combine the content and the shape of lists.

Acknowledgments. Special thanks go to Roman Manevich for his support during the implementation of TVAL+. This work was partially supported by the SNF project “Verification-Driven Inference of Contracts”.

References

1. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker blast. *STTT* 9(5-6), 505–525 (2007)
2. Beyer, D., Henzinger, T.A., Théoduloz, G.: Lazy Shape Analysis. In: Ball, T., Jones, R.B. (eds.) *CAV 2006*. LNCS, vol. 4144, pp. 532–546. Springer, Heidelberg (2006)
3. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: Abstract Domains for Automated Reasoning about List-Manipulating Programs with Infinite Data. In: Kuncak, V., Rybalchenko, A. (eds.) *VMCAI 2012*. LNCS, vol. 7148, pp. 1–22. Springer, Heidelberg (2012)
4. Costantini, G., Ferrara, P., Cortesi, A.: Static Analysis of String Values. In: Qin, S., Qiu, Z. (eds.) *ICFEM 2011*. LNCS, vol. 6991, pp. 505–521. Springer, Heidelberg (2011)

5. Cousot, P.: The calculational design of a generic abstract interpreter. In: *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam (1999)
6. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: *Proceedings of POPL 1977*. ACM Press (1977)
7. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: *Proceedings of POPL 1979*. ACM Press (1979)
8. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: *Proceedings of POPL 1978*. ACM Press (1978)
9. Fähndrich, M., Logozzo, F.: Static Contract Checking with Abstract Interpretation. In: Beckert, B., Marché, C. (eds.) *FoVeOOS 2010*. LNCS, vol. 6528, pp. 10–30. Springer, Heidelberg (2011)
10. Ferrara, P.: *Checkmate*: a generic static analyzer of java multithreaded programs. In: *Proceedings of SEFM 2009*. IEEE Computer Society Press (2009)
11. Ferrara, P.: Static Type Analysis of Pattern Matching by Abstract Interpretation. In: Hatcliff, J., Zucca, E. (eds.) *FMOODS 2010, Part II*. LNCS, vol. 6117, pp. 186–200. Springer, Heidelberg (2010)
12. Ferrara, P., Müller, P.: Automatic Inference of Access Permissions. In: Kuncak, V., Rybalchenko, A. (eds.) *VMCAI 2012*. LNCS, vol. 7148, pp. 202–218. Springer, Heidelberg (2012)
13. Gabi, D.: *Disjunction on demand*. Master thesis, ETH Zürich (2011)
14. Gopan, D., DiMaio, F., Dor, N., Reps, T., Sagiv, M.: Numeric Domains with Summarized Dimensions. In: Jensen, K., Podelski, A. (eds.) *TACAS 2004*. LNCS, vol. 2988, pp. 512–529. Springer, Heidelberg (2004)
15. Gopan, D., Reps, T.W., Sagiv, M.: A framework for numeric analysis of array operations. In: *Proceedings of POPL 2005*. ACM Press (2005)
16. Gulwani, S., Tiwari, A.: Combining abstract interpreters. In: *Proceedings of PLDI 2006*. ACM Press (2006)
17. Jeannet, B., Miné, A.: *APRON: A Library of Numerical Abstract Domains for Static Analysis*. In: Bouajjani, A., Maler, O. (eds.) *CAV 2009*. LNCS, vol. 5643, pp. 661–667. Springer, Heidelberg (2009)
18. Lev-Ami, T., Sagiv, M.: *TVLA: A framework for kleene logic based static analyses*. Master’s thesis, Tel Aviv University (2000)
19. Magill, S., Berdine, J., Clarke, E., Cook, B.: Arithmetic Strengthening for Shape Analysis. In: Riis Nielson, H., Filé, G. (eds.) *SAS 2007*. LNCS, vol. 4634, pp. 419–436. Springer, Heidelberg (2007)
20. Mauborgne, L., Rival, X.: Trace Partitioning in Abstract Interpretation Based Static Analyzers. In: Sagiv, M. (ed.) *ESOP 2005*. LNCS, vol. 3444, pp. 5–20. Springer, Heidelberg (2005)
21. McCloskey, B., Reps, T., Sagiv, M.: Statically Inferring Complex Heap, Array, and Numeric Invariants. In: Cousot, R., Martel, M. (eds.) *SAS 2010*. LNCS, vol. 6337, pp. 71–99. Springer, Heidelberg (2010)
22. Miné, A.: *The octagon abstract domain*. Higher-Order and Symbolic Computation (2006)
23. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. *TOPLAS* 24(3), 217–298 (2002)
24. Spoto, F.: *JULIA: A Generic Static Analyser for the Java Bytecode*. In: *Proceedings of FTfJP 2004* (2005)
25. Zanioli, M., Ferrara, P., Cortesi, A.: *SAILS: static analysis of information leakage with Sample*. In: *Proceedings of SAC 2012*. ACM Press (2012)

A Systematic Approach to Atomicity Decomposition in Event-B

Asieh Salehi Fathabadi¹, Michael Butler², and Abdolbaghi Rezazadeh³

University of Southampton, UK
{asf08r,mjb,ra3}@ecs.soton.ac.uk

Abstract. Event-B is a state-based formal method that supports a refinement process in which an abstract model is elaborated towards an implementation in a step-wise manner. One weakness of Event-B is that control flow between events is typically modelled implicitly via variables and event guards. While this fits well with Event-B refinement, it can make models involving sequencing of events more difficult to specify and understand than if control flow was explicitly specified. New events may be introduced in Event-B refinement and these are often used to decompose the atomicity of an abstract event into a series of steps. A second weakness of Event-B is that there is no explicit link between such new events that represent a step in the decomposition of atomicity and the abstract event to which they contribute. To address these weaknesses, atomicity decomposition diagrams support the explicit modelling of control flow and refinement relationships for new events. In previous work, the atomicity decomposition approach has been evaluated manually in the development of two large case studies, a multi media protocol and a spacecraft sub-system. The evaluation results helped us to develop a systematic definition of the atomicity decomposition approach, and to develop a tool supporting the approach. In this paper we outline this systematic definition of the approach, the tool that supports it and evaluate the contribution that the tool makes.

1 Introduction

The Event-B formal method [1] is an evolution of classical B [2]. Event-B is proven to be applicable in a wide range of domains, including distributed algorithms, railway systems and electronic circuits. The Event-B modelling language has a simple notation and structure. States of a system are defined by variables and state changes of a system are defined by guarded actions, also called events. The basic specification construct is a machine that is comprised of variables and events. Event-B supports refinement [3] in which an abstract model is elaborated towards an implementation in a step-wise manner. During refinement steps a model can be modified and enriched.

One weakness of Event-B is that control flow between events is typically modelled implicitly. Since the Event-B language is a state-based language, ordering between several events can only be modelled in event guards which include conditions on state variables. Because Event-B is also used to model systems with

rich control flow properties, it has been observed that explicit control flow specification is beneficial [4], [5].

A second weakness of Event-B is that all refinement relationships between refinement events and the abstract events are not explicit. Refinement in Event-B can consist of introducing new events. Although the refinement process in Event-B provides a flexible approach to modelling, it is not able to explicitly show the relationships between abstract events and new events introduced during a refinement step.

To address these weaknesses, the atomicity decomposition approach [6] addresses the explicit control flow modelling and explicit refinement relationships representation. It provides a graphical notation to structure the refinement process and to illustrate the explicit ordering between events of a model. The atomicity decomposition graphical notation contains tree structured diagrams based on Jackson Structure Diagrams (JSD) [7]. Semantics are given to an atomicity decomposition diagram by generating an Event-B model from it.

In the rest of this paper, “AD” refers to Atomicity Decomposition. The steps carried in our research are presented in Figure 1. AD is first introduced by Butler [6] (step 1). It has been observed that methodological support for AD was weak. So we decided to evaluate and enhance the existing AD approach from [6]. For this reason we manually applied AD to two sizeable case studies, a multi media protocol and a space craft system (step 2). The first case study, the multi media protocol [8], contains requirements to establish, modify and close a media channel between two endpoints for transferring multi media data. Second case study is based on a space craft system called BepiColombo [9]. Developments of both these case studies involving manual translation of AD diagrams to Event-B have been published in [10] and [11] respectively. Insights gained from these case studies, enable us to define a formal description of the AD language (ADL) and formal translation rules from AD diagrams to Event-B (step 3). Based on the ADL and translation rule descriptions, we have developed the AD tool support, as a plug-in for the Event-B tool-set, called Rodin (step 4). Our AD tool support, can automatically generate Event-B models from AD diagrams. And finally we re-developed the case study models using the provided AD tool support (step 5).



Fig. 1. Road Map

The contribution of this paper is to present ADL and translation rules from AD diagrams to the Event-B language, covering steps 3, 4 and 5 of Figure 1. We also outline the development of the AD tool and the technologies that were used in this tool development. One of our objectives in this paper is to assess

how application of translation rules, makes the automatic models of case studies more consistent and systematic, compares with with the previous manual ones.

The paper is structured as follows: Section 2 outlines the Event-B method, atomicity decomposition approach, related work and an overview of case studies requirements; Section 3 contains the ADL description and definitions of translation rules; Section 4 presents the tool developed to support AD; In Section 5 we evaluate how the AD tool has helped us to enhance the development of case studies in a more consistent and systematic way compared with the manual development; finally Section 6 concludes.

2 Background and Related Work

2.1 Event-B

The Event-B formal method [1], [12] has evolved from classical B [2] and action systems [13]. Event-B is used in modelling and verifying consistency of models. The modelling language is based on set theory and first order logic.

A model in Event-B consists of several *Contexts* and *Machines*. Contexts contain the static part (types and constants) of a model while machines contain the dynamic part (variables and events). Contexts provide axiomatic properties of an Event-B model, whereas Machines provide behavioural properties of an Event-B model. A context can be “extended” by other contexts and “referenced” by machines. A machine can be “refined” by other machines and can reference contexts.

Building a model in Event-B usually starts with an abstract level, and continues in successive refinement levels. The abstract model provides a simple view of the system, focusing on the main purposes of the system. Details are added gradually to the abstract model during refinement levels. In Event-B, refinement is used to introduce new functionality or add details of current functionality. One of the important features of Event-B refinement is the ability to introduce new events in a refinement level. From a given machine, *Machine1*, a new machine, *Machine2*, can be built as a refinement of *Machine1*. In this case, *Machine1* is called an abstraction of *Machine2*, and *Machine2* will said to be a concrete version of *Machine1*.

Rodin [14] is an Eclipse-based tool for formal modelling and proving in Event-B. Rodin has an open platform, and is an extensible and adaptable modelling tool. We have taken the advantage of the extensibility feature of the Rodin to develop a tool support for the AD approach.

2.2 Atomicity Decomposition Approach

Although refinement in Event-B provides a flexible approach to modelling, it has the weakness that we cannot explicitly represent the relationships between abstract events and new events which are introduced in a refinement level. The AD approach addresses this limitation. The idea is to augment Event-B refinement with a graphical notation that is capable of representing the relationships

between abstract and concrete events explicitly. Using the AD approach has another advantage which is that we can represent event ordering explicitly. Figure 2 illustrates these two features of the AD graphical notation.

Assume machine $M1$ on the left hand side of Figure 2, refines some machine $M0$ which contains the abstract specification of $AbstractEvent$. The machine $M1$ encodes its control flow (ordering between $Event1$ and $Event2$) via guards on the events. This control flow is made explicit in the AD diagram presented in the right hand side. This diagram explicitly illustrates that the effect achieved by $AbstractEvent$ at the abstract level, machine $M0$, is realized at the refined level, machine $M1$, by occurrence of $Event1$ followed by $Event2$. The ordering of the leaf events is always from left to right (this is based on JSD diagrams [7]). The solid line indicates that $Event2$ refines $AbstractEvent$ while the dashed line indicates that $Event1$ is a new event which refines $skip$. In the Event-B model of machine $M1$ on the left hand side, $Event1$ does not have any explicit connection with $AbstractEvent$, but the diagram indicates that we break the atomicity of $AbstractEvent$ into two sub-events in the refinement.

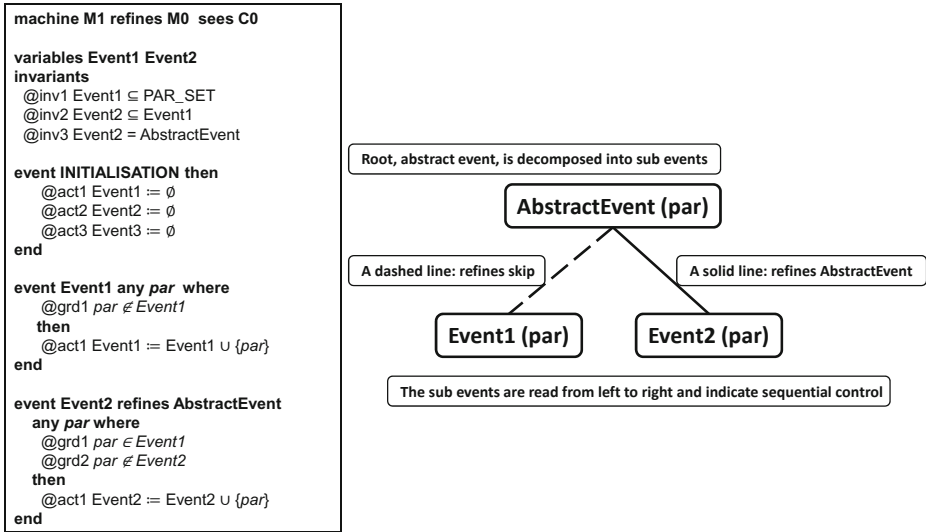


Fig. 2. Atomicity Decomposition Diagram

The parameter par in the diagram indicates that we are modelling multiple instances of $AbstractEvent$ and its sub-events. Events associated with different values of par may be interleaved thus modelling interleaved execution of multiple processes. The effect of an event with parameter par , is to add the value of par to a set control variable with the same name as the event, i.e., $par \in Event1$ means that $Event1$ has occurred with value par . The use of a set means that the same event can occur multiple times with different values for par . The guard of an event with value par specifies that the event has not already occurred for

value *par* but has occurred for the previous event, e.g., the guard of *Event2* says that *Event1* has occurred and *Event2* has not occurred for value *par*.

2.3 Related Work

The desire to explicitly model control flow is not restricted to Event-B. To address this issue usually a combination of two formal methods are suggested. A good example of such an approach is Circus [15] combining CSP [16] and Z [17]. The combination of CSP and classical B [2] has also been investigated in [4] and [18].

To provide explicit control flow for an Event-B model, a combination of two formal methods is presented in [19] which is based on using CSP alongside Event-B. As presented in Section 2.2, control flow can only be implicitly modelled in state variables and event guards in Event-B. On the other hand CSP is a process-based formalism, which explicitly supports specifying control flow via processes.

UML-B [20] provides a “UML-like” graphical front-end for Event-B. It adds support for class-oriented and state machine modelling. State machines provide us with a graphical notation to explicitly define event sequencing. Events are represented by transitions on a state machine, and control flow is specified by defining the source and target state of each transition.

Another method to explicitly define control flow properties of an Event-B model is suggested in [21]. This method extends Event-B models with expressions, called flows, defining event ordering. Flows are written in a language resembling those in process algebra.

All the techniques outlined in this section only deal with explicit event sequencing; they do not support the explicit refinement relationships provided by atomicity decomposition diagrams. The atomicity decomposition approach provides a graphical front-end to Event-B along with other features such as supporting explicit event sequencing and expressing refinement relationships between abstract and refinement events. An extra feature of the AD approach is that the graphical front-end of it can provide an overall visualisation of the refinement structure, which is not supported by any of techniques outlined above.

2.4 Overview of Case Studies

This section outlines an overview of our case study systems, a multi media protocol [8] and a space craft system based on BepiColombo [9].

Multi Media Protocol. This case study specifies a protocol for establishing, modifying and closing a media channel. A media channel is established for transferring multi-media data. There are three phases in the protocol: establish, modify and close. In the modification phase some properties of the established channel can be modified, such as the codec used for data encoding.

It is worth to compare our approach to the multi media protocol with the approach taken by Zave and Cheung [8]. Zave and Cheung present Promela

models of the behaviour of each end of the protocol and use the Spin model checker to verify that these models satisfy certain safety and liveness properties. In our approach with Event-B, we start with a more global view of the intention of the protocol and then use atomicity decomposition to arrive at models that have similar levels of detail to the Promela models.

Space Craft System. Exploration of the planet Mercury is the main goal of the BepiColombo mission [9]. One of the BepiColombo subsystems consists of a core and four devices. The core and the control software are responsible for controlling the power of devices and their operation states and to handle TeleCommand (TC) and TeleMessage (TM) communications. In our work, we treat a part of the BepiColombo system related to the management of TC and TM communications. The core software (CSW) plays a management role over the devices. CSW is responsible for communication with Earth on one hand and with the devices on the other hand. Here is the summary of the system requirements:

- A TeleCommand (*TC*) is received by the core from Earth.
- The CSW checks the syntax of the received *TC*.
- Further semantic checking has to be carried out on the syntactically validated *TC*. If the *TC* contains a message for one of the devices, it has to be sent to the device for semantic checking, otherwise the semantic checking is carried out in the core.
- For each valid *TC* a control TeleMessage (*TM*) is generated and sent to Earth.

3 AD Language and Translation Rules

3.1 Atomicity Decomposition Language

To describe the AD language (ADL) syntax, we adopted Augmented Backus-Naur Form (ABNF) [22]. ABNF is a metalanguage based on Backus-Naur Form (BNF).

An excerpt of the ADL syntax, describing a single AD diagram, is presented in Figure 3. This description is only a subset of the full ADL. It only includes three of the AD constructors which are used in our case studies and are explained later in the following sections. There are other AD constructors which are not presented in this paper because of space limitation¹.

A flow, in Figure 3, refers to a single atomicity decomposition. To describe the type of a line (solid/dashed), we consider a boolean property, called “ref”. When a sub-event refines the abstract event (solid line), “ref” is one; otherwise “ref” is zero. Considering Figure 3, the ABNF of ADL may be described informally as follows:

¹ The full list of the AD constructors is presented in the PhD thesis of the first author of this paper: <http://eprints.soton.ac.uk/340357/>

| | | |
|-------------|---|---|
| flow | = | "flow" (name, *par) (1*child (ref)) |
| child | = | "leaf" (name) / constructor |
| constructor | = | "loop" ("leaf" (name)) / "xor" (2* "leaf" (name)) / "one" (par) ("leaf" (name)) |

Fig. 3. Syntax of the AD Language (ADL)

- A flow consists of a name, zero or more parameters, followed by one or more children. Each child of a flow has a “*ref*” property.
- A child is either a “*leaf*” with a name, or a constructor.
- A constructor is either a “*loop*” with one leaf as its child or a “*xor*” with two or more leaves or an “*one*” with a parameter, followed by one leaf.

3.2 Translation Rules

Semantics are given to an AD diagram by generating an Event-B model from it, based on some translation rules. In this section, we discuss these translation rules. Here, due to space limitation, we only present translation rules that are used in our two case studies. [\[2\]](#) The initial AD diagrammatic notation in [\[6\]](#) has been extended with some AD constructors. Three of them, *loop*, *xor* and *one*, used in our case study developments are introduced here.

The main syntactic elements of an Event-B machine are variables, invariants, guards and actions. The encoding of AD diagrams in Event-B uses a collection of Event-B syntactic patterns such as typing invariants, sequencing invariants, partitioning invariants, disabling guards, sequencing guards and leaf actions. Our translation scheme defines a separate rule for each of these syntactic patterns. [Figure 4](#) outlines the full list of translation rules used in this paper. Each translation rule defines a transformation from an AD source element to an Event-B destination element. Note that for each AD element usually there is more than one applicable translation rule. We explain the role of each translation rule using snippets taken from the case studies. We first explain the rules related to sequencing of events, then the rules for the *loop* constructor, a solid leaf, the *xor* and *one* constructors.

Sequencing Rules. As discussed in [Section 2.2](#), one major feature of AD diagrams is to explicitly represent sequencing between events. To illustrate this concept, we have taken a part of the most abstract level diagrams of the Bepi-Colombo system, presented in the upper level of [Figure 5](#). In the most abstract diagram, the name of the system appears in an oval as the root node, and the names of the most abstract events appear in the leaves in an order from left to right. This diagram illustrates the scenario when a *TC* is received by the core, *ReceiveTC* event, and then it is validated by *TC_Validation_Ok* event.

² The full set of translation rules is presented in the PhD thesis of the first author of this paper: <http://eprints.soton.ac.uk/340357/>

| | |
|---|---|
| TR1: leaf → leaf variable | TR8: loop → <i>loop</i> guard |
| TR2: first leaf → typing invariant | TR9: solid leaf → gluing invariant |
| TR3: non-first leaf → sequencing invariant | TR10: solid leaf → refining event |
| TR4: leaf → non-refining event | TR11: solid xor → partitioning invariant |
| TR5: leaf → disabling guard | TR12: xor → <i>xor</i> guard |
| TR6: non-first leaf → sequencing guard | TR13: one → cardinality invariant |
| TR7: leaf → leaf action | TR14: one → <i>one</i> guard |

Fig. 4. Translation Rules

The arrows in Figure 5 indicate the application of translation rules. For example, the TR1 arrow from the *ReceiveTC* leaf in the diagram to the *ReceiveTC* variable in the Event-B model shows that the application of TR1 rule to each source leaf produces a variable in the Event-B model. The generated variables are used to control the flow of the leaf events.

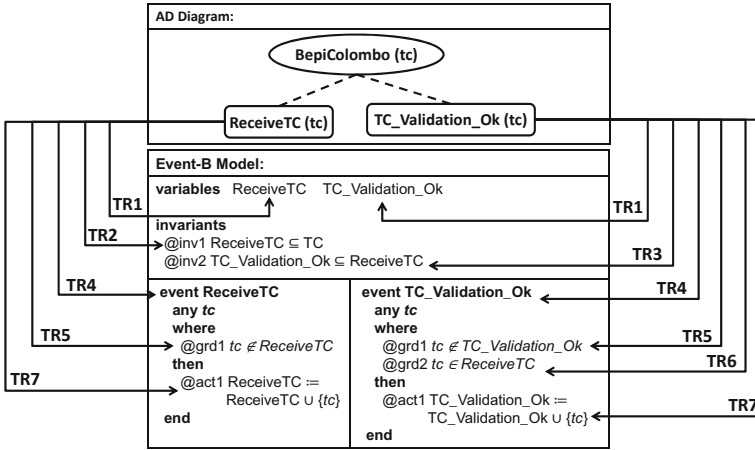


Fig. 5. The Most Abstract Level Model, BepiColombo System

Application of TR2 to the first leaf produces an invariant which defines the type of the leaf variable. Application of TR3 to the second leaf produces an invariant which describes the sequencing constraint between two leaf events. The sequencing invariant describes the second leaf variable as a subset of the previous leaf variable, since the second leaf event is allowed to execute only after execution of its previous leaf event.

In the most abstract diagram, since all leaves illustrate the most abstract events, there is no solid line. For each leaf with a dashed line, TR4 generates a non-refining event. The parameter of the leaf is transformed to the event parameter. For each leaf, TR5 generates a disabling guard, which describes that the leaf event has not executed for the same instance of the parameter before. For each non-first event, like *TC_Validation_Ok* here, another guard is needed

to make sure that the previous event has been executed for the parameter value before; this translation is carried out via TR6. Finally TR7 adds an action for each leaf, which disables the corresponding leaf event for a specific parameter value.

Translation rules (TR1-TR7) that are outlined in Figure 4 and applied in this section, are only applicable to leaf nodes and encode sequencing collectively. We discuss the rest of rules in the following sections.

Loop Constructor. The loop constructor is used to model zero or more executions of a leaf. Figure 6 presents the most abstract AD diagram of the multi media protocol which contains the loop constructor as its second child. The diagram states that first a media channel is established, then it can be modified zero or more times and finally it is closed. Considering Figure 6, there is no variable generated for the leaf connected to the loop constructor, since we do not need to record the loop event execution. The event after the loop event can execute after execution of the event before the loop event (in the case of zero executions of the loop event).

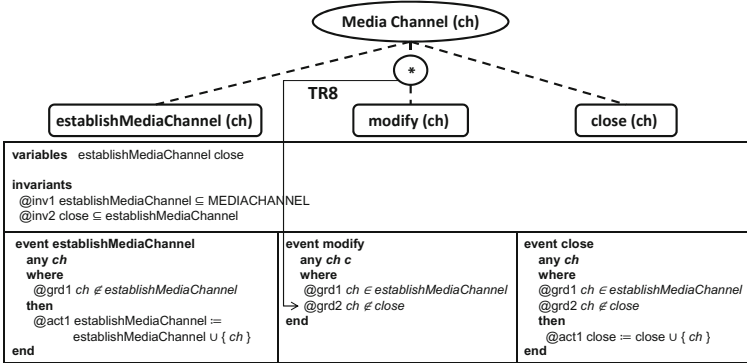


Fig. 6. The Most Abstract Level, Multi Media Protocol

The loop event can execute several times *before* execution of next event. TR8 transforms the loop constructor to a guard in the loop event, *modify*. This guard checks that the event after loop, *close*, has not executed before, for the intended channel. The other Event-B elements in Figure 6 are generated via TR1-TR7 which have been described in the previous section (Figure 4).

Solid Line. The abstract atomic *TC_Validation_Ok* event in Figure 5, is decomposed to three sub-events in a refinement level. Figure 7 presents the AD diagram of this decomposition. Validating a received *TC* is not atomic. It is done in two steps, checking the syntax, in *TCCheck_Ok* event, and the semantics, in *TCExecute_Ok* event, of a received *TC*. After syntax and semantics checks, in the third step, *TCExecOk_ReplyCtrlTM*, a control *TM* is produced and sent back to Earth.

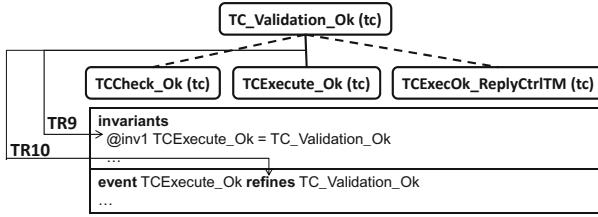


Fig. 7. The AD Diagram of *TC_Validation_Ok*, BepiColombo System

Considering Figure 7, a solid line in an AD diagram has two effects. First, it is translated to an invariant which connects the abstract variable to the refinement variable (TR9); this is called a gluing invariant. Second, it is translated to an event which refines the abstract event in the root node (TR10).

xor Constructor. Exclusive choice between two or more events is introduced to the AD diagram with a new constructor called *xor*. An application of the *xor* constructor in BepiColombo development is presented in Figure 8. A *TC* either belongs to the core or the device and not both of them. The figure illustrates a further level of refinement where the atomicity of semantics checking event, *TCEXecute_Ok*, is decomposed to an exclusive choice between two sub-events; *TCCoreExecute_Ok* event checks the semantics of a *TC* which belongs to the core and *TCDeviceExecute_Ok* event checks the semantics of a *TC* which belongs to the device.

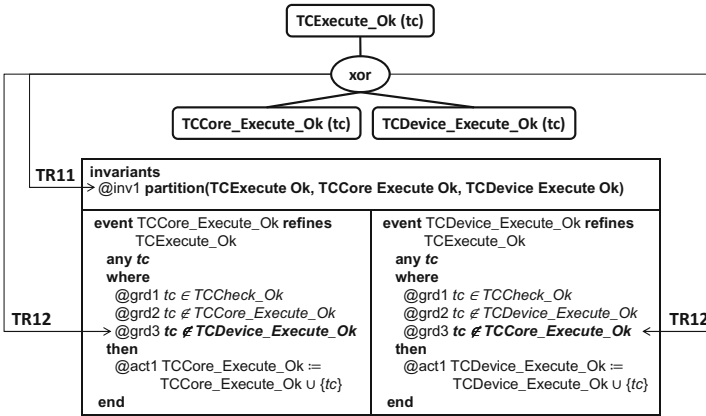


Fig. 8. The AD Diagram of *TC_Execute_Ok*, BepiColombo System

xor sub-leaves inherit the type of their line (solid/dashed) from the *xor* constructor. Considering Figure 8, the *xor* constructor is connected to the root node with a solid line, therefore both *xor* sub-leaves are connected with solid lines and refine the abstract event in the root node.

There are two translation rules for the *xor* constructor. First the *xor* constructor is transformed to the partitioning invariant (TR11), which ensures exclusivity of execution. The *partition* operator in Event-B is defined as follows:

$$\text{partition}(E_0, E_1, \dots, E_n) \equiv (E_0 = E_1 \cup \dots \cup E_n) \wedge (i \neq j \Rightarrow E_i \cap E_j = \emptyset)$$

The generated partitioning invariant first describes the relationship between the abstract variable and the refinement variables:

$$TCExecute_Ok = TCCoreExecute_Ok \cup TCDeviceExecute_Ok.$$

Second it described the mutually exclusive property of the the *xor* sub-events:

$$TCCoreExecute_Ok \cap TCDeviceExecute_Ok = \emptyset.$$

The second translation rule (TR12) generates a guard for each *xor* sub-event. This guard enforces the exclusiveness property of *xor* sub-events. The guard in each *xor* sub-event checks that the other *xor* sub-events have not occurred for the intended value of the *TC*.

one Constructor. The *one* constructor specifies execution of an event for exactly one instance value of a new parameter. An application of the *one* constructor in BepiColombo development is presented in Figure 9. Figure 9 illustrates that the *TCExecOk_ReplyCtrlTM* event is decomposed to produce exactly one *TM*, in the *TCExecOk_ProcessCtrlTM* event, followed by the completion action, *TCExecOk_CompleteCtrlTM* event.

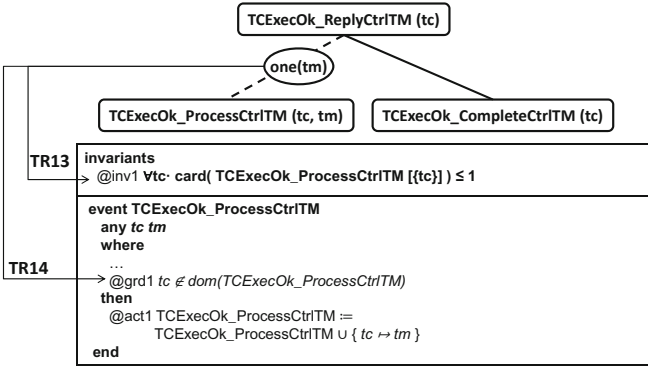


Fig. 9. The AD Diagram of *TCExecOk_ReplyCtrlTM*, BepiColombo System

As presented in Figure 9, the *one* constructor adds a new parameter, the *tm* parameter, to its sub-event, *TCExecOk_ProcessCtrlTM*. For each validated *tc*, exactly one control *tm* should be processed. To enforce this constraint, the *one* constructor is translated to an invariant and a guard. TR13 generates an invariant which defines the *one* constructor property describing that for each *tc*, the cardinality of the set of processed *tms* is at most one. And TR14 generates a guard to make sure that the *one* sub-event has not executed for the same value of intended *tc* before.

There are two more constructors, *all* and *some*, which adds a new parameter to their sub-events. The *all* constructor specifies execution of an event for all

instance values of a new parameter. And the *some* constructor specifies execution of an event for some instance values of a new parameter. In this paper, we skip defining them in depth.

Using the formal description of ADL, presented in Figure 3, the translation rules outlined in Figure 4, can be defined formally. For instance, the formal description of TR1 is presented in Figure 10. The left-hand box contains the AD element description that is transformed to the right-hand box containing the description of the Event-B element. In the case of TR1, each leaf (not the loop leaf) is transformed to a variable (with the same name as the leaf) in the generated Event-B model.

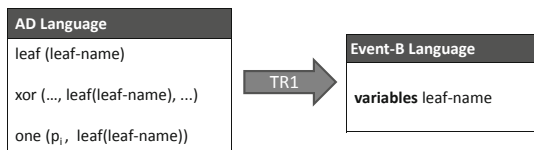


Fig. 10. TR1 Definition

4 Tool Support

Eclipse [23], is a multi-language software development environment comprising an integrated development environment (IDE) and an extensible plug-in system. The Rodin platform is an Eclipse-based IDE for Event-B and is further extendable with plug-ins. By taking advantage of the extensibility feature of the Rodin platform for Event-B, we have developed a plug-in as tool support for the AD approach. The AD plug-in helps developers to build Event-B models more easily, since the AD plug-in addresses automatic generation of the Event-B models in term of control flows and refinement relationships. The AD plug-in allows users to define the AD diagram; then the AD diagram is automatically transformed to an Event-B model.

The development architecture is briefly presented in Figure 11. We define the ADL specification in an EMF (Eclipse Modelling Framework) [24] meta-model, called the source meta-model, and then the source meta-model is transformed to the Event-B EMF meta-model as the target meta-model. Currently AD diagrams are build as an EMF model, included in an Event-B machine. However in the future we plan to develop a graphical environment for the plug-in. The transformation is done using the Epsilon Transformation Language (ETL) [25]. ETL is a rule-based model-to-model transformation language.



Fig. 11. AD Tool Support Architecture

The ETL rule for TR1 (presented in Section 3.2) is as follow:

```
rule Leaf2Varibale
  transform l : Source!Leaf
  to v : Target!Variable {
    v.name := l.name; }
```

This rule transforms a leaf from the ADL meta-model (as the source meta-model) to a variable in the Event-B meta-model (as the target meta-model). In the body of rule the name of the target component (variable) is assigned to the name of the source component (leaf).

5 Evaluation

Our AD tool addresses automatic generation of control flow in Event-B modelling. Moreover using the AD plug-in to create the Event-B model of a system, ensures a consistent encoding of the AD diagrams in a systematic way. The manually generated Event-B models are less systematic and less consistent, since at the time of developing them our experience of AD applications were not enough. The versions of the case studies reported in this paper are referred to as automatically generated models. We applied the tool to the two case studies and compared the automatic models with the manual models, reported in our earlier work. There are some differences between the automatic models and the manual models, of which some of the more notable ones are described in this Section.

5.1 Naming Protocol

In the automatic Event-B models (like Figure 5), each control variable has the same name as the corresponding event name. Whereas in the manual Event-B models, there was no specific naming protocol for variables name. Providing a unique naming protocol helps to understand the model more easily, and can help to track the ordering between events more easily.

5.2 Alternative Approaches of Control Flow Modelling in Event-B

There are different approaches to model control flow in Event-B. In the automatic Event-B model, we adopted the subset approach to model ordering between sequential events. Considering Figure 5, the second control variable is a subset of the first one (*inv1*). The alternative way is disjoint sets. The Event-B model

| | |
|---|--|
| <pre>event ReceiveTC any tc where @grd1 tc ∈ ReceiveTC then @act1 ReceiveTC := ReceiveTC ∪ {tc} end</pre> | <pre>event TC_Validation_Ok any tc where @grd1 tc ∈ ReceiveTC then @act1 ReceiveTC := ReceiveTC / {tc} @act2 TC_Validation_Ok := TC_Validation_Ok ∪ {tc} end</pre> |
|---|--|

Fig. 12. Disjoint Sets in the Most Abstract Level, BepiColombo System

of disjoint sets for the diagram in Figure 5 is presented in Figure 12. In this way the parameter tc is removed from $ReceiveTC$ set variable in the body of $TC_Validation_Ok$ event.

One of the advantages of using the subset relationships in the Event-B models is that the subset relationships between the control variables that represent different states of the model can be specified in the invariants of the model. Considering Figure 5, invariant $inv2$ specifies the ordering relationship between control variables. This ensures that the orderings are upheld in the Event-B model more strongly than if specified only in the event guards. Moreover, having disjoint set variables would not allow us to model some of the constructors in a simple way as subset variables provide.

5.3 A Merged Guard versus Separate Guards

Considering the automatic Event-B model in Figure 5, there is a separate guard for each predicate ($grd1$ and $grd2$ in the $TC_Validation_Ok$ event). These separate guards are generated as a result of different translation rules (TR5 and TR6 respectively). Whereas in the manual Event-B model, we modeled all of the precondition predicates in a single guard. For instance, guards of $TC_Validation_Ok$ event in Figure 5, can be merged as a single guard ($tc \in ReceiveTC \setminus TC_Validation_Ok$).

To verify the correctness and consistency of an Event-B model, some proof obligations are generated by Rodin provers. Some of the generated proof obligations are related to the guards verification. Proving such proof obligations generated for the manual Event-B models needs more effort comparing to the proof obligations generated for the automatic Event-B models, since the corresponding separated guards are simpler predicates compared to a merged guard.

6 Conclusion

In the previous publications we have demonstrated how the atomicity decomposition (AD) approach provides a means of introducing explicit flow control into Event-B development process. In this paper, we have presented the formal description of the atomicity decomposition language (ADL) and translation rules from the ADL to the Event-B language. We have developed a tool, supporting the atomicity decomposition methodology; the tool support is developed as a plug-in for the Event-B tool-set, Rodin. A brief description of AD tool development has been illustrated. Using translation rules developed in the AD tool, has helped us to develop the models of the previous case studies in an automatic way. Compared to the previous manual models of the case studies, the recent automatic models are more consistent and systematic. Some aspects of this improvement have been outlined.

The current AD tool does not provide a graphical environment of AD diagrams. Instead an AD diagram is represented as an EMF model that is manipulated using an EMF structure editor. We consider developing a graphical

environment of AD diagrams as future work. Also future work is needed in order to improve the ADL and translation rules. For this reason, further applications of the AD approach using the AD tool are being undertaken.

Acknowledgement. This work is partly supported by the EU research project ICT 214158 DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity) www.deploy-project.eu.

References

1. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
2. Abrial, J.-R.: The B-book: Assigning Programs to Meanings. Cambridge University Press (1996)
3. Abrial, J.-R.: Refinement, Decomposition and Instantiation of Discrete Models. In: Abstract State Machines, pp. 17–40 (2005)
4. Butler, M.: csp2B: A Practical Approach to Combining CSP and B. In: Formal Aspects of Computing, vol. 12, pp. 934–5043 (2000) ISSN 0934-5043
5. Iliasov, A.: On Event-B and Control Flow. Technical Report, School of Computing Science, Newcastle University (2009)
6. Butler, M.: Decomposition Structures for Event-B. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 20–38. Springer, Heidelberg (2009)
7. Jackson, M.A.: System Development. Prentice-Hall, Englewood Cliffs (1983)
8. Zave, P., Cheung, E.: Compositional Control of IP Media. IEEE Trans. Software Eng. 35(1), 46–66 (2009)
9. ESA Media Center, Space Science. Factsheet: Bepicolombo (2008), <http://www.esa.int/esaSC>
10. Fathabadi, A.S., Butler, M.: Applying Event-B Atomicity Decomposition to a Multi Media Protocol. In: FMCO Formal Methods for Components and Objects, pp. 89–104 (2010)
11. Fathabadi, A.S., Rezazadeh, A., Butler, M.: Applying Atomicity and Model Decomposition to a Space Craft System in Event-B. In: NASA Formal Methods, pp. 328–342 (2011)
12. Metayer, C., Abrial, J.-R., Voisin, L.: Event-B language. RODIN Project Deliverable 3.2 (2005), <http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf>
13. Back, R.-J., Kurki-Suonio, R.: Distributed Cooperation with Action Systems. ACM Trans. Program. Lang. Syst., 513–554 (1988)
14. Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: An Open Toolset for Modelling and Reasoning in Event-B. In: STTT, vol. 12, pp. 447–466 (2010)
15. Woodcock, J., Cavalcanti, A.: The semantics of $\$circus\$$. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) B 2002 and ZB 2002. LNCS, vol. 2272, pp. 184–203. Springer, Heidelberg (2002)
16. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985) ISBN 0-13-153289-8
17. Davies, J., Woodcock, J.: Using Z: Specification, Refinement and Proof. Prentice Hall International Series in Computer Science (1996) ISBN 0-13-948472-8

18. Schneider, S., Treharne, H.: Verifying Controlled Components. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 87–107. Springer, Heidelberg (2004)
19. Schneider, S., Treharne, H., Wehrheim, H.: A CSP Approach to Control in Event-B. In: Méry, D., Merz, S. (eds.) IFM 2010. LNCS, vol. 6396, pp. 260–274. Springer, Heidelberg (2010)
20. Said, M.Y., Butler, M., Snook, C.: Language and Tool Support for Class and State Machine Refinement in UML-B. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 579–595. Springer, Heidelberg (2009)
21. Iliasov, A.: Tutorial on the Flow plugin for Event-B. In: Workshop on B Dissemination (WOBD) Satellite event of SBMF, Natal, Brazil (2010)
22. Crocker, D., Overell, P.: Augmented BNF for Syntax Specifications: ABNF. STD 68, RFC 5234 (2008)
23. Eclipse (Online), <http://www.eclipse.org>
24. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Part of the Eclipse Series series. Published by Addison-Wesley Professional (2008)
25. Kolovos, D., Rose, L., Paige, R.: The Epsilon Book (2008), <http://www.eclipse.org/gmt/epsilon/doc/book>

Compositional Reasoning about Shared Futures^{*}

Crystal Chang Din, Johan Dovland, and Olaf Owe

Department of Informatics, University of Oslo, Norway
{crystal.d, johand, olaf}@ifi.uio.no

Abstract. Distributed and concurrent object-oriented systems are difficult to analyze due to the complexity of their concurrency, communication, and synchronization mechanisms. The *future mechanism* extends the traditional method call communication model by facilitating sharing of references to futures. By assigning method call result values to futures, third party objects may pick up these values. This may reduce the time spent waiting for replies in a distributed environment. However, futures add a level of complexity to program analysis, as the program semantics becomes more involved.

This paper presents a model for asynchronously communicating objects, where return values from method calls are handled by futures. The model facilitates invariant specifications over the locally visible communication history of each object. Compositional reasoning is supported, as each object may be specified and verified independently of its environment. A kernel object-oriented language with futures inspired by the ABS modeling language is considered. A compositional proof system for this language is presented, formulated within dynamic logic.

1 Introduction

Distributed systems play an essential role in society today. However, quality assurance of distributed systems is non-trivial since they depend on unpredictable factors, such as different processing speeds of independent components. Therefore, it is highly challenging to test such distributed systems after deployment under different relevant conditions. These challenges motivates frameworks combining precise modeling and analysis with suitable tool support. In particular, *compositional verification systems* allow the different components to be analyzed independently from their surrounding components.

Object orientation is the leading framework for concurrent and distributed systems, recommended by the RM-ODP [23]. However, method-based communication between concurrent units may cause busy-waiting, as in the case of remote and synchronous method invocation, e.g., Java RMI [2]. Concurrent objects communicating by *asynchronous method calls* have been proposed as a promising framework to combine object-orientation and distribution in a natural manner. Each concurrent object encapsulates its own state and processor, and

^{*} Supported by the EU project FP7-231620 *HATS: Highly Adaptable and Trustworthy Software using Formal Models* (<http://www.hats-project.eu>) and COST Action IC0701.

internal interference is avoided as at most one process is executing on an object at a time. Asynchronous method calls allow the caller to continue with its own activity without blocking while waiting for the reply, and a method call leads to a new process on the called object. The notion of *futures* [8,20,27,31] improves this setting by providing a decoupling of the process invoking a method and the process reading the returned value. By sharing *future identities*, the caller enables other objects to wait for method results. However, futures complicate program analysis since programs become more involved compared to semantics with traditional method calls, and in particular local reasoning is a challenge.

The execution of a distributed system can be represented by its *communication history* or *trace*; i.e., the sequence of observable communication events between system components [10,22]. At any point in time the communication history abstractly captures the system state [13,14]. In fact, traces are used in semantics for full abstraction results (e.g., [5,24]). The *local history* of an object reflects the communication visible to that object, i.e., between the object and its surroundings. A system may be specified by the finite initial segments of its communication histories, and a *history invariant* is a predicate which holds for all finite sequences in the set of possible histories, expressing safety properties [7].

In this work we consider a kernel object-oriented language, where futures are used to manage return values of method calls. Objects are concurrent and communicate asynchronously. We formalize object communication by a four event operational semantics, capturing shared futures, where each event is visible to only one object. Consequently, the local histories of two different objects share no common events, and history invariants can be established independently for each object. We present a *dynamic logic* proof system for class verification, facilitating independent reasoning about each class. A verified *class invariant* can be instantiated to each object of that class, resulting in an invariant over the local history of the object. Modularity is achieved as the independently derived history invariants can be composed to form *global* system specifications. Global history consistency is captured by a notion of history *wellformedness*. The formalization of object communication extends previous work [17] which considered concurrent objects and asynchronous communication, but without futures.

Paper overview. Sect. 2 presents a core language with shared futures. The communication model is presented in Sect. 3, and Sect. 4 defines the operational semantics. Sect. 5 presents the compositional reasoning system, and Sect. 6 contains related work and concludes the paper.

2 A Core Language with Shared Futures

A *future* is a placeholder for the return value of a method call. Each future has an unique identity which is *generated* when a method is invoked. The future is *resolved* upon method termination, by placing the return value of the method in the future. Thus, unlike the traditional method call mechanism, the callee does not send the return value directly back to the caller. However, the caller may keep a *reference* to the future, allowing the caller to *fetch* the future value

| | |
|--|----------------------------|
| $Cl ::= \mathbf{class} C([T\ cp]^*) \{[T\ w\ [:=\ e]^?]^* M^*\}$ | class definition |
| $M ::= T\ m([T\ x]^*) \{\mathbf{var} [T\ x]^? s ; \mathbf{return} e\}$ | method definition |
| $T ::= C \mid \mathbf{Int} \mid \mathbf{Bool} \mid \mathbf{String} \mid \mathbf{Void} \mid \mathbf{Fut} \langle T \rangle$ | types |
| $v ::= x \mid w$ | variables (local or field) |
| $e ::= \mathbf{null} \mid \mathbf{this} \mid v \mid cp \mid f(\bar{e})$ | pure expressions |
| $s ::= v := e \mid fr := v!m(\bar{e}) \mid v := e?$ | statements |
| $\quad \mid \mathbf{skip} \mid \mathbf{if} e \mathbf{then} s \mathbf{else} s \mathbf{fi} \mid s ; s$ | |

Fig. 1. Core language syntax, with C class name, cp formal class parameter, m method name, w fields, x method parameter or local variable, and where fr is a future variable. We let $[]^*$ and $[]^?$ denote repeated and optional parts, respectively, and \bar{e} is a (possibly empty) expression list. Expressions e and functions f are side-effect free.

once resolved. References to futures may be shared between objects, e.g., by passing them as parameters. After achieving a future reference, this means that third party objects may fetch the future value. Thus, the future value may be fetched several times, possibly by different objects. In this manner, shared futures provide an efficient way to distribute method call results to a number of objects.

For the purposes of this paper, we consider a core object-oriented language with futures, presented in Fig 1. It includes basic statements for first order futures, inspired by *ABS* [21]. Methods are organized in classes in a standard manner. A class C takes a list of formal parameters \bar{cp} , and defines fields \bar{w} and methods \bar{M} . There is read-only access to the parameters \bar{cp} . A method definition has the form $m(\bar{x})\{\mathbf{var} \bar{y}; s; \mathbf{return} e\}$, ignoring type information, where \bar{x} is the list of parameters, \bar{y} an optional list of *method-local variables*, s is a sequence of statements, and the value of e is returned upon termination.

A future variable fr is declared by $\mathbf{Fut} \langle T \rangle fr$, indicating that fr may refer to futures which may contain values of type T . The call statement $fr := x!m(\bar{e})$ invokes the method m on object x with input values \bar{e} . The identity of the generated future is assigned to fr , and the calling process continues execution without waiting for fr to become resolved. The query statement $v := fr?$ is used to fetch the value of a future. The statement blocks until fr is resolved, and then assigns the value contained in fr to v . The language contains additional statements for assignment, **skip**, conditionals, and sequential composition.

We assume that call and query statements are well-typed. If x refers to an object where m is defined with no input values and return type \mathbf{Int} , the following is well-typed: $\mathbf{Fut} \langle \mathbf{Int} \rangle fr; \mathbf{Int} v; fr := x!m(); v := fr?$.

Class instances are concurrent, encapsulating their own state and processor. Each method invoked on the object leads to a new process, and at most one process is executing on an object at a time. Object communication is *asynchronous*, as there is no explicit transfer of control between the caller and the callee. The core language ignores *ABS* features that are orthogonal to shared futures, including interface encapsulation, object creation, local synchronous calls, and internal scheduling of processes by means of cooperative multitasking. We refer to the report version of this paper for a treatment of these issues [18].

```

class Adm(Prof p) {
  String req() {var String cv, Fut<String> mb, Fut<Fut<String>> fr;
    fr := p!ask(); mb:= fr?; cv := mb?;
    ... //register cv
    return cv}}
class Prof(Stud s) {
  Fut<String> ask() {Fut<String> mb; mb := s!getCV(); return mb}}
class Stud(String cv) { String getCV() {return cv}}

```

Fig. 2. A simple example with futures

Example. In order to illustrate the usage of futures, we consider the classes in Fig. 2. The figure provides an implementation of a small system containing an administrator, a professor, and a PhD student hired by the professor. The administrator knows the professor, but does not know the student. In order to get a CV of the student, the administrator sends a request to the professor. The professor suggests a mailbox where the student can submit his CV directly to the administrator. Technically, the mailbox is implemented by a future `mb`. In order to acquire the CV, the administrator invokes the method `ask` on the professor which returns a future reference. The administrator may then query the future directly, and the future is resolved once the student submits a CV. In this manner, the professor does not have to wait for the CV from the student and then forward the result to the administrator. The future identity generated by the professor is the one received by the administrator, which means that the CV received by the administrator is the same as the one submitted by the student.

3 Observable Behavior

In this section we describe a communication model for concurrent objects communicating by means of asynchronous message passing and futures. The model is defined in terms of the *observable* communication between objects in the system. We consider how the execution of an object may be described by different *communication events* which reflect the observable interaction between the object and its environment. The observable behavior of a system is described by communication histories over observable events [10, 22].

3.1 Communication Events

Since message passing is asynchronous, we consider separate events for method invocation, reacting upon a method call, resolving a future, and for fetching the value of a future. Each event is observable to only one object, which is the one that *generates* the event. The events generated by a method call cycle is depicted in Fig. 3. The object o calls a method m on object o' with input values \bar{e} and where u denotes the future identity. An invocation message is sent from o to o' when the method is invoked. This is reflected by the *invocation event*

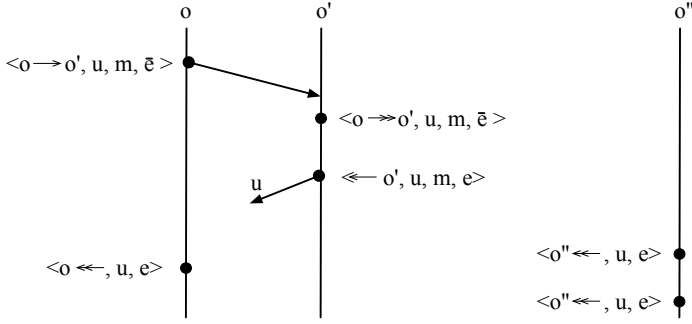


Fig. 3. A method call cycle: object o calls a method m on object o' with future u . The events on the left-hand side are visible to o , those in the middle are visible to o' , and the ones on the right-hand side are visible to o'' . There is an arbitrary delay between message receiving and reaction.

$\langle o \rightarrow o', u, m, \bar{e} \rangle$ generated by o . An invocation reaction event $\langle o \rightarrow o', u, m, \bar{e} \rangle$ is generated by o' once the method starts execution. When the method terminates, the object o' generates the future event $\langle \leftarrow o', u, m, e \rangle$. This event reflects that u is resolved with return value e . The fetching event $\langle o \leftarrow, u, e \rangle$ is generated by o when o fetches the value of the resolved future. Since future identities may be passed to other objects, e.g. o'' , this object may also fetch the future value, reflected by the event $\langle o'' \leftarrow, u, e \rangle$, generated by o'' . Let type `Mid` include all method names, and let `Data` be the supertype of all values occurring as actual parameters, including future identities `Fid` and object identities `Oid`.

Definition 1. (Events) Let $caller, callee, receiver : \text{Oid}$, $future : \text{Fid}$, $method : \text{Mid}$, $args : \text{List}[\text{Data}]$, and $result : \text{Data}$. Communication events Ev include:

- Invocation events $\langle caller \rightarrow callee, future, method, args \rangle$, generated by caller.
- Invocation reaction events $\langle caller \rightarrow callee, future, method, args \rangle$, generated by callee.
- Future events $\langle \leftarrow callee, future, method, result \rangle$, generated by callee.
- Fetching events $\langle receiver \leftarrow, future, result \rangle$, generated by receiver

Events may be decomposed by functions. For instance, $_ . \text{result} : \text{Ev} \rightarrow \text{Data}$ is well-defined for future and fetching events, e.g., $\langle \leftarrow o', u, m, e \rangle . \text{result} = e$.

For a method invocation with future u , the ordering of events depicted in Fig. 3 is described by the following regular expression (using \cdot for sequential composition of events)

$$\langle o \rightarrow o', u, m, \bar{e} \rangle \cdot \langle o \rightarrow o', u, m, \bar{e} \rangle \cdot \langle \leftarrow o', u, m, e \rangle [\cdot \langle _ \leftarrow, u, e \rangle]^*$$

for some fixed o, o', m, \bar{e}, e , and where $_$ denotes an arbitrary value. This implies that the result value may be read several times, each time with the same value, namely that given in the preceding future event.

3.2 Communication Histories

The execution of a system up to present time may be described by its history of observable events, defined as a sequence. A sequence over some type T is constructed by the empty sequence ε and the right append function $_ \cdot _ : \text{Seq}[T] \times T \rightarrow \text{Seq}[T]$ (where “ $_$ ” indicates an argument position). The choice of constructors gives rise to generate inductive function definitions, in the style of [14]. Projection, $_ / _ : \text{Seq}[T] \times \text{Set}[T] \rightarrow \text{Seq}[T]$ is defined inductively by $\varepsilon / s \triangleq \varepsilon$ and $(a \cdot x) / s \triangleq \mathbf{if} \ x \in s \ \mathbf{then} \ (a/s) \cdot x \ \mathbf{else} \ a/s \ \mathbf{fi}$, for $a : \text{Seq}[T]$, $x : T$, and $s : \text{Set}[T]$, restricting a to the elements in s . A *communication history* for a set S of objects is defined as a sequence of events generated by the objects in S . We say that a history is *global* if S includes all objects in the system.

Definition 2. (Communication histories) *The communication history h of a system of objects S is a sequence of type $\text{Seq}[\text{Ev}]$, such that each event in h is generated by an object in S .*

We observe that the *local history* of a single object o is achieved by restricting S to the single object, i.e., the history contains only elements generated by o . For a history h , we let h/o abbreviate the projection of h to the events generated by o . Since each event is generated by only one object, it follows that the local histories of two different objects are disjoint.

Definition 3. (Local histories) *For a global history h and an object o , the projection h/o is the local history of o .*

4 Operational Semantics

Rewriting logic [28] is a logical and semantic framework in which concurrent and distributed systems can be specified in an object-oriented style. Unbounded data structures and user-defined data types are defined in this framework by means of equational specifications. Rewriting logic extends membership equational logic with *rewrite rules*, so that in a *rewrite theory*, the *dynamic* behavior of a system is specified as a set of rules on top of its *static* part, defined by a set of equations. Informally, a *labeled conditional rewrite rule* is a transition $l : t \longrightarrow t' \ \mathbf{if} \ \mathit{cond}$, where l is a *label*, t and t' are terms over typed variables and function symbols of given arities, and cond is a condition that must hold for the transition to take place. Rewrite rules are used to specify local transitions in a system, from a state fragment that matches the pattern t , to another state fragment that is an instance of the pattern t' . Rules are selected nondeterministic if there are at least two rule instantiations with left-hand sides matching overlapping fragments of a term. Concurrent rewriting is possible if the fragments are non-overlapping. Furthermore, matching is made modulo the properties of the function symbols that appear in the rewrite rule, like associativity, commutativity, identity (ACI), which introduces further nondeterminism. The Maude tools [11] allow simulation, state exploration, reachability analysis, and LTL model checking of rewriting logic specifications. The state of a concurrent object system is captured by

a *configuration*, which is a multiset of *units* such as objects and messages, and other relevant system parts, which in our case includes futures. Concurrency is then supported in the framework by allowing concurrent application of rules when there are non-overlapping matches of left-hand sides.

4.1 Operational Rules

For our purpose, a configuration is a multiset of (concurrent) objects, classes, messages, futures, as well as a representation of the global history. We use blank-space as the multiset constructor, allowing ACI pattern matching. Objects have the form **object**($Id : o, A$) where o is the unique identity of the object and A is a set of semantic attributes, including

| | |
|------------|---|
| $Cl : c$ | the class c of the object, |
| $Pr : s$ | the remaining code s of the active process, |
| $Lvar : l$ | the local state l of the active process, including method parameters and the implicit future identity destiny , |
| $Flds : a$ | the state a of the fields, including class parameters, |
| $Cnt : n$ | a counter n used to generate future identities, |
| $Mtd : m$ | the name m of the current method. |

Similarly, classes have the form **class**($Id : c, Mtds : d$). Here, d is a multiset of method definitions of the form (m, \bar{p}, l, s) , where m is the method name, \bar{p} is the list of parameters, l contains the local variables (including default values), and s is the code. The history is represented by a unit **hist**(h) where h is finite sequence of events (initially empty). Messages have the form of invocation events as described above. At last, a future unit is of the form **fut**($Id : u, Val : v$) where u is the future identity and v is its value. Remark that a system configuration contains exactly one history. The history is included to define the interleaving semantics upon which we derive our history-based reasoning formalism.

The operational rules are summarized in Fig. 4. Method invocation is captured by the rule **call**. The generated future identity (o, n) is globally unique (assuming the *next* function is producing locally unique values). The future identity is generated by this rule, but no future unit. If there is no active process in an object, denoted $Pr : empty$, a method call is selected for execution by rule **method**. The invocation message is removed from the configuration by this rule, and the future identity of the call is assigned to the implicit parameter **destiny**. Method execution is completed by rule **return**, and a future value is fetched by rule **query**. A future unit appears in the configuration when resolved by rule **return**, which means that a query statement blocks until the future is resolved. Remark that rule **query** does not remove the future unit from the configuration, which allows several processes to fetch the value of the same future. The given language fragment may be extended with object creation and constructs for inter object process control and suspension, e.g., by using the *ABS* approach of [17].


```

skip:   object(Id : o, Pr : (skip; s)) → object(Id : o, Pr : s)

assign : object(Id : o, Pr : (v := e; s), Lvar : l, FlDs : a)
        →
        if v in Lvar then object(Id : o, Pr : s, Lvar : l[v := eval(e, (a; l))], FlDs : a)
        else object(Id : o, Pr : s, Lvar : l, FlDs : a[v := eval(e, (a; l))])

call :   hist(h) object(Id : o, Pr : (fr := v!m(ē); s), Lvar : l, FlDs : a, Cnt : n)
        →
        MSG hist(h · MSG)
        object(Id : o, Pr : (fr := (o, n); s), Lvar : l, FlDs : a, Cnt : next(n))

method : ⟨o' → o, u, m, v̄⟩ hist(h) class(Id : c, Mtds : (q (m, p̄, l, s)))
        →
        object(Id : o, Cl : c, Pr : empty, FlDs : a)
        →
        hist(h · ⟨o' → o, u, m, v̄⟩) class(Id : c, Mtds : (q (m, p̄, l, s)))
        object(Id : o, Cl : c, Pr : s, Lvar : l[p̄ := v̄][destiny := u], FlDs : a, Mtd : m)

return : hist(h) object(Id : o, Pr : return e, Lvar : l, FlDs : a, Mtd : m)
        →
        hist(h · ⟨← o, eval(destiny, l), m, eval(e, (a; l))⟩)
        fut(Id : eval(destiny, l), Val : eval(e, (a; l)))
        object(Id : o, Pr : empty, FlDs : a)

query : hist(h) fut(Id : u, Val : d) object(Id : o, Pr : (v := e?; s), Lvar : l, FlDs : a)
        →
        hist(h · ⟨o ←, u, d⟩) fut(Id : u, Val : d)
        object(Id : o, Pr : (v := d; s), Lvar : l, FlDs : a)
        if eval(e, (a; l)) = u
    
```

Fig. 4. Operational rules, using the standard rewriting logic convention that irrelevant attributes may be omitted in a rule. Variables are denoted by single characters (the uniform naming convention is left implicit), $(a; l)$ represents the total object state, and $a[v := d]$ is the state a updated by binding the variable v to the data value d . The $eval$ function evaluates an expression in a given state, and in is used for testing domain membership. In rule `call`, `MSG` denotes $\langle o \rightarrow eval(v, (a; l)), (o, n), m, eval(\bar{e}, (a; l)) \rangle$, where (o, n) is the generated future identity.

4.2 Semantic Properties

Given the operational semantics, we provide a notion of wellformedness for global histories. We first introduce some notation and functions used in defining wellformed histories. For sequences a and b , let $a \mathbf{ew} x$ denote that x is the last element of a , $agree(a)$ denote that all elements (if any) are equal, and $a \leq b$ denote that a is a prefix of b . Let $[x_1, x_2, \dots, x_i]$ denote the sequence of x_1, x_2, \dots, x_i for $i > 0$ (allowing repeated parts $[...]^*$). Functions for event decomposition are lifted to sequences in the standard way, ignoring events for which the decomposition is not defined, e.g., $_ .result : \text{Seq}[\text{Ev}] \rightarrow \text{Seq}[\text{Data}]$. The function $fid : \text{Seq}[\text{Ev}] \rightarrow \text{Set}[\text{Fid}]$ extracts future identities from a history:

$$\begin{aligned}
 fid(\varepsilon) &\triangleq \emptyset & fid(h \cdot \gamma) &\triangleq fid(h) \cup fid(\gamma) \\
 fid(\langle o \rightarrow o', u, m, \bar{e} \rangle) &\triangleq \{u\} & fid(\langle o' \rightarrow o, u, m, \bar{e} \rangle) &\triangleq \{u\} \cup fid(\bar{e}) \\
 fid(\langle \leftarrow o, u, m, e \rangle) &\triangleq \emptyset & fid(\langle o \leftarrow, u, e \rangle) &\triangleq fid(e)
 \end{aligned}$$

where $\gamma : \text{Ev}$, and $fid(\bar{e})$ returns the set of future identities occurring in the expression list \bar{e} . For a global history h , the function $fid(h)$ returns all future

identities on h , and for a local history h/o , the function $fid(h/o)$ returns the futures generated by o or received as parameters. At last, h/u abbreviates the projection of history h to the set $\{\gamma \mid \gamma.future = u\}$, i.e., all events with future u .

Definition 4. (Wellformed histories) Let $h : Seq[Ev]$ be a history of a global object system S . The wellformedness predicate $wf : Seq[Ev] \rightarrow Bool$ is defined by:

$$\begin{aligned} wf(\varepsilon) &\triangleq true \\ wf(h \cdot \langle o \rightarrow o', u, m, \bar{e} \rangle) &\triangleq wf(h) \wedge o \neq null \wedge u \notin fid(h) \\ wf(h \cdot \langle o' \rightarrow o, u, m, \bar{e} \rangle) &\triangleq wf(h) \wedge o \neq null \wedge h/u = [\langle o' \rightarrow o, u, m, \bar{e} \rangle] \\ wf(h \cdot \langle \leftarrow o, u, m, e \rangle) &\triangleq wf(h) \wedge h/u \text{ \textbf{ew} } \langle _ \rightarrow o, u, m, _ \rangle \\ wf(h \cdot \langle o \leftarrow, u, e \rangle) &\triangleq wf(h) \wedge u \in fid(h/o) \wedge agree((h/u).result) \cdot e \end{aligned}$$

It follows directly that a wellformed global history satisfies the communication order pictured in Fig. 3, i.e.,

$$\forall u. \exists o, o', m, \bar{e}, e. \\ h/u \leq [\langle o' \rightarrow o, u, m, \bar{e} \rangle, \langle o' \rightarrow o, u, m, \bar{e} \rangle, \langle \leftarrow o, u, m, e \rangle, [\langle _ \leftarrow, u, e \rangle]^*]$$

We can prove that the operational semantics guarantees wellformedness:

Lemma 1. *The global history h of a global object system S obtained by the given operational semantics, is wellformed, $wf(h)$.*

This lemma follows by induction over the number of rule applications, assuming that object identifiers are unique (and not `null`). Wellformedness of a local history for an object o , denoted $wf_o(h/o)$, is defined as in Def. 4, except that the last conjunct of the case $\langle o' \rightarrow o, u, m, \bar{e} \rangle$ only holds for self calls, i.e., where o and o' are equal. For local wellformedness, the conjunct is therefore weakened to $o = o' \Rightarrow h/u = [\langle o' \rightarrow o, u, m, \bar{e} \rangle]$. If h is a wellformed global history, it follows immediately that each projection h/o is locally wellformed, i.e., $wf_o(h/o)$ holds.

5 Program Verification

The communication history abstractly captures the system state at any point in time [13, 14]. Partial correctness properties of a system may thereby be specified by finite initial segments of its communication histories. A *history invariant* is a predicate over the communication history, which holds for all finite sequences in the (prefix-closed) set of possible histories, expressing safety properties [7]. In this section we present a framework for compositional reasoning about object systems, establishing an invariant over the global history from invariants over the local histories of each object. Since the local object histories are disjoint with our four event semantics, it is possible to reason locally about each object. In particular, the history updates of the operational semantics affect the local history of the active object only, and can be treated simply as an assignment to the local history. The local history is not effected by the environment, and interference-free reasoning is then possible. Correspondingly, the reasoning framework consists of two parts: A proof system for local (class-based) reasoning, and a rule for composition of object specifications.

5.1 Local Reasoning

Pre- and postconditions to method definitions are in our setting used to establish a *class invariant*. The class invariant must hold after initialization of all class instances and must be maintained by all methods, serving as a contract for the different methods: A method implements its part of the contract by ensuring that the invariant holds upon termination, assuming that it holds when the method starts execution. A class invariant establishes a *relationship between the internal state and the observable behavior of class instances*. The internal state reflects the values of the fields, and the observable behavior is expressed as potential communication histories. A *user-provided invariant* $I_C(\overline{w}, \mathcal{H})$ for a class C is a predicate over the fields \overline{w} , the read-only parameters \overline{cp} and **this**, in addition to the local history \mathcal{H} which is a sequence of events generated by **this**. The proof system for class-based verification is formulated within *dynamic logic* as used by the KeY framework [9], facilitating class invariant verification by considering each method independently. The dynamic logic formulation suggests that the proof system is suitable for an implementation in the KeY framework.

Dynamic logic provides a structured way to describe program behavior by an integration of programs and assertions within a single language. The formula $\psi \Rightarrow [s]\phi$ express partial correctness properties: if statement s is executed in a state where ψ holds and the execution terminates, then ϕ holds in the final state. The formula is verified by a symbolic execution of s , where state modifications are handled by the *update* mechanism [9]. A dynamic formula $[v := e; s]\phi$, i.e., where an assignment is the first statement, reduces to $\{v := e\}[s]\phi$, where $\{v := e\}$ is an update. We assume that expressions e can be evaluated within the assertion language. Updates can only be *applied* on formulas without programs, which means that updates on a formula $[s]\phi$ are accumulated and *delayed* until the symbolic execution of s is complete. Update application $\{v := t\}e$, on an expression e , evaluates to the substitution e_t^v , replacing all free occurrences of v in e by t . The parallel update $\{v_1 := e_1 || \dots || v_n := e_n\}$, for disjoint variables v_1, \dots, v_n , represents an accumulated update, and the application of a parallel update leads to a simultaneous substitution. A *sequent* $\psi_1, \dots, \psi_n \vdash \phi_1, \dots, \phi_m$ contains assumptions ψ_1, \dots, ψ_n , and formulas ϕ_1, \dots, ϕ_m to be proved. The sequent is *valid* if at least one formula ϕ_i follows from the assumptions, and it can be interpreted as $\psi_1 \wedge \dots \wedge \psi_n \Rightarrow \phi_1 \vee \dots \vee \phi_m$.

In order to verify a class invariant $I_C(\overline{w}, \mathcal{H})$, we must prove that the invariant is maintained by all method definitions in C , assuming wellformedness of the local history. For each method definition $m(\overline{x})\{s; \mathbf{return} e\}$ in C , this amounts to a proof of the sequent:

$$\vdash (wf_{\mathbf{this}}(\mathcal{H}) \wedge I_C(\overline{w}, \mathcal{H})) \Rightarrow [\mathcal{H} := \mathcal{H} \cdot \langle \mathbf{caller} \rightarrow \mathbf{this}, \mathbf{destiny}, m, \overline{x} \rangle; \\ s; \mathcal{H} := \mathcal{H} \cdot \langle \leftarrow \mathbf{this}, \mathbf{destiny}, m, e \rangle](wf_{\mathbf{this}}(\mathcal{H}) \Rightarrow I_C(\overline{w}, \mathcal{H}))$$

Here, the method body is extended with a statement for extending the history with the invocation reaction event, and the **return** statement is treated as a history extension. Dynamic logic rules for method invocation and future query can be found in Fig. 5. When invoking a method, the update in the premise of

$$\text{invoc} \frac{\vdash \forall u. \{ \mathcal{H} := \mathcal{H} \cdot \langle \text{this} \rightarrow o, u, m, \bar{e} \rangle \mid \text{fr} := u \} [s] \phi}{\vdash [\text{fr} = o!m(\bar{e}); s] \phi}$$

$$\text{fetch} \frac{\vdash \forall v'. \{ \mathcal{H} := \mathcal{H} \cdot \langle \text{this} \leftarrow, e, v' \rangle \mid v := v' \} [s] \phi}{\vdash [v := e?; s] \phi}$$

Fig. 5. Dynamic logic rules for method invocation and future query

rule `invoc` captures the history extension and the generation of a future identity u . Similarly, the update in rule `fetch` captures the history extension and the assignment of a fresh value to v , where the wellformedness assumptions ensure that all values received from the same future are equal. The soundness of these rules is straightforward with respect to the semantics in Sec. 4. Assignments are analyzed as explained above, and rules for `skip` and conditionals are standard. We refer to Din et al. for further details [18].

5.2 Compositional Reasoning

The class invariant for some class C is a predicate over class parameters \bar{cP} , fields \bar{w} , and the local projection of possible global histories h . History invariants for instances of C , expressed as a predicate over the local history, can be derived from the class invariant. For an instance o of C with actual parameter values \bar{e} , the history invariant $I_{o:C(\bar{e})}(h/o)$ is defined by hiding the internal state \bar{w} and instantiating `this` and the class parameters:

$$I_{o:C(\bar{e})}(h/o) \triangleq \exists \bar{w}. I_C(\bar{w}, h/o)_{o, \bar{e}}^{\text{this}, \bar{cP}}$$

The history invariant $I_S(h)$ for a system S is then given by combining the history invariants of the composed objects:

$$I_S(h) \triangleq \text{wf}(h) \bigwedge_{(o:C(\bar{e})) \in S} I_{o:C(\bar{e})}(h/o)$$

The wellformedness property serves as a connection between the local histories. Note that the system invariant is obtained directly from the history invariants of the composed objects, without any restrictions on the local reasoning, since the local histories are disjoint. This ensures compositional reasoning. The composition rule is similar to [17], which also considers dynamically created objects.

5.3 Example

In this example we consider object systems based on the classes found in Fig. 2. Assume that the global system consists of the objects $a : \text{Adm}$, $p : \text{Prof}$, $s : \text{Stud}$, and $m : \text{Main}$, where the only visible activity of m is that it invokes `req` on the administrator. The semantics may lead to several global histories for this system,

depending on the interleaving of the different object activities. One global history H caused by a call to `req` is as follows:

$$\begin{aligned} & \langle m \rightarrow a, u_1, \text{req}, \varepsilon \rangle, \langle m \rightarrow a, u_1, \text{req}, \varepsilon \rangle, \langle a \rightarrow p, u_2, \text{ask}, \varepsilon \rangle, \langle a \rightarrow p, u_2, \text{ask}, \varepsilon \rangle, \\ & \langle p \rightarrow s, u_3, \text{getCV}, \varepsilon \rangle, \langle \leftarrow p, u_2, \text{ask}, u_3 \rangle, \langle p \rightarrow s, u_3, \text{getCV}, \varepsilon \rangle, \langle a \leftarrow, u_2, u_3 \rangle, \\ & \langle \leftarrow s, u_3, \text{getCV}, cv \rangle, \langle a \leftarrow, u_3, cv \rangle, \langle \leftarrow a, u_1, \text{req}, cv \rangle \end{aligned}$$

It follows that the CV received by the administrator must be the same as the one submitted by the student. In addition, the future identity u_3 generated by the professor is the one received by the administrator. We may derive these properties within the proof system from the following class invariants:

$$\begin{aligned} I_{Stud(cv)}(\mathcal{H}) & \triangleq \mathcal{H} \leq [\langle c \rightarrow \text{this}, u, \text{getCV}, \varepsilon \rangle, \langle \leftarrow \text{this}, u, \text{getCV}, cv \rangle . \mathbf{some} \ c, u]^* \\ I_{Prof(s)}(\mathcal{H}) & \triangleq \mathcal{H} \leq [\langle c \rightarrow \text{this}, u, \text{ask}, \varepsilon \rangle, \langle \text{this} \rightarrow s, u', \text{getCV}, \varepsilon \rangle, \\ & \quad \langle \leftarrow \text{this}, u, \text{ask}, u' \rangle . \mathbf{some} \ c, u, u']^* \\ I_{Adm(p)}(\mathcal{H}) & \triangleq \mathcal{H} \leq [\langle c \rightarrow \text{this}, d, \text{req}, \varepsilon \rangle, \langle \text{this} \rightarrow p, u, \text{ask}, \varepsilon \rangle, \\ & \quad \langle \text{this} \leftarrow, u, u' \rangle, \langle \text{this} \leftarrow, u', cv \rangle, \langle \leftarrow \text{this}, d, \text{req}, cv \rangle . \mathbf{some} \ c, d, u, u', cv]^* \end{aligned}$$

letting $h \leq p^*$ express that h is a prefix of a repeated pattern p where additional variables occurring in p (after **some**) may change for each repetition. These invariants are straightforwardly verified in the above proof system. The corresponding object invariants for $s : Stud(cv)$, $p : Prof(s)$, and $a : Adm(p)$ are obtained by substituting actual values for `this` and class parameters:

$$\begin{aligned} I_{s:Stud(cv)}(h/s) & \triangleq h/s \leq [\langle _ \rightarrow s, u, \text{getCV}, \varepsilon \rangle, \langle \leftarrow s, u, \text{getCV}, cv \rangle . \mathbf{some} \ u]^* \\ I_{p:Prof(s)}(h/p) & \triangleq h/p \leq [\langle _ \rightarrow p, u, \text{ask}, \varepsilon \rangle, \langle p \rightarrow s, u', \text{getCV}, \varepsilon \rangle, \\ & \quad \langle \leftarrow p, u, \text{ask}, u' \rangle . \mathbf{some} \ u, u']^* \\ I_{a:Adm(p)}(h/a) & \triangleq h/a \leq [\langle _ \rightarrow a, d, \text{req}, \varepsilon \rangle, \langle a \rightarrow p, u, \text{ask}, \varepsilon \rangle, \\ & \quad \langle a \leftarrow, u, u' \rangle, \langle a \leftarrow, u', cv \rangle, \langle \leftarrow a, d, \text{req}, cv \rangle . \mathbf{some} \ d, u, u', cv]^* \end{aligned}$$

The global invariant of a system S with the objects, $s : Stud(cv)$, $p : Prof(s)$, $a : Adm(p)$, and $m : Main(a)$ is then

$$I_S(h) \triangleq \text{wf}(h) \wedge I_{a:Adm(p)}(h/a) \wedge I_{p:Prof(s)}(h/p) \wedge I_{s:Stud(cv)}(h/s) \wedge I_{m:Main(a)}(h/m)$$

where wellformedness allows us to relate the different object histories. From this global invariant we may derive that the CV received by the administrator must be the same as the one submitted by the student.

As a special case, we consider a system where main invokes the `req` method once, i.e. $I_{m:Main(a)}(h/m) \triangleq h/m \leq [\langle m \rightarrow a, u, \text{req}, \varepsilon \rangle . \mathbf{some} \ u]$. History wellformedness then ensures that the cycles defined by the remaining invariants are repeated at most once, and that variables in the patterns are connected, i.e., the future u in $I_{m:Main(a)}$ is identical to the future d in $I_{a:Adm(p)}$. The global invariant then reduces to the following:

$$\begin{aligned} I_S(h) & \triangleq \text{wf}(h) \wedge h/m \leq [\langle m \rightarrow a, u_1, \text{req}, \varepsilon \rangle \wedge \\ & \quad h/a \leq [\langle m \rightarrow a, u_1, \text{req}, \varepsilon \rangle, \langle a \rightarrow p, u_2, \text{ask}, \varepsilon \rangle, \langle a \leftarrow, u_2, u_3 \rangle, \\ & \quad \langle a \leftarrow, u_3, cv \rangle, \langle \leftarrow a, u_1, \text{req}, cv \rangle] \wedge \\ & \quad h/p \leq [\langle a \rightarrow p, u_2, \text{ask}, \varepsilon \rangle, \langle p \rightarrow s, u_3, \text{getCV}, \varepsilon \rangle, \langle \leftarrow p, u_2, \text{ask}, u_3 \rangle] \wedge \\ & \quad h/s \leq [\langle p \rightarrow s, u_3, \text{getCV}, \varepsilon \rangle, \langle \leftarrow s, u_3, \text{getCV}, cv \rangle] \end{aligned}$$

This invariant allows a number of global histories, depending on the interleaving of the activities in the different objects. The history H presented first in this section satisfies the invariant, and represents one particular interleaving.

6 Related Work and Conclusion

Models for asynchronous communication without futures have been explored for process calculi with buffered channels [22], for agents with message-based communication [1], for method-based communication [4], and in particular for Java [3]. Behavioral reasoning about distributed and object-oriented systems is challenging, due to the combination of concurrency, compositionality, and object orientation. Moreover, the gap in reasoning complexity between sequential and distributed, object-oriented systems makes tool-based verification difficult in practice. A survey of these challenges can be found in [6]. The present approach follows the line of work based on communication histories to model object communication events in a distributed setting [10,12,22]. Objects are concurrent and interact solely by method calls and futures, and remote access to object fields are forbidden.

By creating unique references for method calls, the *label* construct of Creol [26] resembles futures, as callers may postpone reading result values. Verification systems capturing Creol labels can be found in [6,19]. However, a label reference is local to the caller, and cannot be shared with other objects. A reasoning system for futures has been presented in [16], using a combination of global and local invariants. Futures are treated as visible objects rather than reflected by events in histories. In contrast to our work, global reasoning is obtained by means of global invariants, and not by compositional rules. Thus the environment of a class must be known at verification time.

A reasoning system for asynchronous methods in ABS without futures is presented in [17], from which we redefine the four-event semantics to reflect actions on shared futures. The semantics gives a clean separation of the activities of the different objects, which leads to disjointness of local histories. Thus, object behavior can be specified in terms of the observable interaction of the current object only. This is essential for obtaining a simple reasoning system. In related approaches, e.g., [6,19], events are visible to more than one object. The local histories must then be updated with the activity of other objects, resulting in more complex reasoning systems. Based on the four-event semantics, we present a compositional reasoning system for distributed, concurrent objects with asynchronous method calls. A class invariant defines a relation between the inner state and the observable communication of instances, and can be verified independently for each class. The class invariant can be instantiated for each object of the class, resulting in a history invariant over the observable behavior of the object. Compositional reasoning is ensured as history invariants may be combined to form global system specifications. The composition rule is similar to [17], which is inspired by previous approaches [29,30].

In order to focus on the future mechanism, this paper considers a core language with shared futures. The report version [18] considers a richer language,

including constructs for object creation and inter-object process control. The verification system is suitable for an implementation within the KeY framework. With support for (semi-)automatic verification, such an implementation will be valuable when developing larger case studies. It is also natural to investigate how our reasoning system would benefit from extending it with rely/guarantee style reasoning. Assumptions about callee behavior may, for instance, be used to express properties of return values. More sophisticated techniques may also be used, e.g., [15,25] adapts rely/guarantee style reasoning to history invariants. However, such techniques requires more complex composition rules. Soundness proofs for local reasoning and object composition rules are left as future work.

References

1. Agha, G., Frølund, S., Kim, W., Panwar, R., Patterson, A., Sturman, D.: Abstraction and modularity mechanisms for concurrent computing. *IEEE Parallel Distributed Technology: Systems Applications* 1(2), 3–14 (1993)
2. Ahern, A., Yoshida, N.: Formalising Java RMI with explicit code mobility. *Theoretical Computer Science* 389(3), 341–410 (2007)
3. Falkner, K., Coddington, P.D., Oudshoorn, M.J.: Implementing asynchronous remote method invocation in Java. In: *Proc. PART 1999* (November 1999)
4. Morandi, B., Bauer, S.S., Meyer, B.: SCOOP – A Contract-Based Concurrent Object-Oriented Programming Model. In: Müller, P. (ed.) *LASER Summer School 2007/2008*. LNCS, vol. 6029, pp. 41–90. Springer, Heidelberg (2010)
5. Ábrahám, E., Grabe, I., Grüner, A., Steffen, M.: Behavioral interface description of an object-oriented language with futures and promises. *Journal of Logic and Algebraic Programming* 78(7), 491–518 (2009)
6. Ahrendt, W., Dylla, M.: A system for compositional verification of asynchronous objects. *Science of Computer Programming* (2010)
7. Alpern, B., Schneider, F.B.: Defining liveness. *IPL* 21(4), 181–185 (1985)
8. Baker Jr., H.G., Hewitt, C.: The incremental garbage collection of processes. In: *Proc. 1977 Symposium on Artificial Intelligence and Programming Languages*, pp. 55–59. ACM (1977)
9. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): *Verification of Object-Oriented Software: The KeY Approach*. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
10. Broy, M., Stølen, K.: *Specification and Development of Interactive Systems*. Monographs in Computer Science. Springer (2001)
11. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: *All About Maude - A High-Performance Logical Framework*. LNCS, vol. 4350. Springer, Heidelberg (2007)
12. Dahl, O.-J.: Can program proving be made practical?. In: *Les Fondements de la Programmation*. Institut de Recherche d’Informatique et d’Automatique, Toulouse, France, pp. 57–114 (1977)
13. Dahl, O.-J.: Object-oriented specifications. In: *Research Directions in Object-Oriented Programming*, pp. 561–576. MIT Press, Cambridge (1987)
14. Dahl, O.-J.: *Verifiable Programming*. International Series in Computer Science. Prentice Hall (1992)
15. Dahl, O.-J., Owe, O.: *Formal methods and the RM-ODP*. Research Report 261, Dept. of Informatics, University of Oslo, Norway (1998)

16. de Boer, F.S., Clarke, D., Johnsen, E.B.: A Complete Guide to the Future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)
17. Din, C.C., Dovland, J., Johnsen, E.B., Owe, O.: Observable behavior of distributed systems: Component reasoning for concurrent objects. *Journal of Logic and Algebraic Programming* 81(3), 227–256 (2012)
18. Din, C.C., Dovland, J., Owe, O.: An approach to compositional reasoning about concurrent objects and futures. Research Report 415, Dept. of Informatics, University of Oslo (2012), <http://folk.uio.no/crystald/rr415.pdf>
19. Dovland, J., Johnsen, E.B., Owe, O.: Verification of concurrent objects with asynchronous method calls. In: Proc. SwSTE 2005, pp. 141–150. IEEE Press (2005)
20. Halstead Jr., R.H.: Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.* 7(4), 501–538 (1985)
21. Full ABS Modeling Framework, Deliverable 1.2 of project FP7-231620 (HATS) (March 2011), <http://www.hats-project.eu>
22. Hoare, C.A.R.: *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall (1985)
23. International Telecommunication Union. *Open Distributed Processing - Reference Model parts 1–4*. Technical report, ISO/IEC, Geneva (1995)
24. Jeffrey, A., Rathke, J.: Java JR: Fully Abstract Trace Semantics for a Core Java Language. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 423–438. Springer, Heidelberg (2005)
25. Johnsen, E.B., Owe, O.: Object-Oriented Specification and Open Distributed Systems. In: Owe, O., Krogdahl, S., Lyche, T. (eds.) *From OO to FM* (Dahl Festschrift). LNCS, vol. 2635, pp. 137–164. Springer, Heidelberg (2004)
26. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling* 6(1), 35–58 (2007)
27. Liskov, B.H., Shriram, L.: Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In: Proc. PLDI 1988. ACM Press (1988)
28. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science* 96, 73–155 (1992)
29. Soundararajan, N.: Axiomatic semantics of communicating sequential processes. *ACM TOPLAS* 6(4), 647–662 (1984)
30. Soundararajan, N.: A proof technique for parallel programs. *Theoretical Computer Science* 31(1-2), 13–29 (1984)
31. Yonezawa, A., Briot, J.-P., Shibayama, E.: Object-oriented concurrent programming in ABCL/1. In: Proc. OOPSLA 1986. *Sigplan Notices*, vol. 21(11), pp. 258–268 (1986)

Verification of Aspectual Composition in Feature-Modeling*

Qinglei Zhang, Ridha Khedri, and Jason Jaskolka

Department of Computing and Software,
McMaster University, Hamilton, Ontario, Canada
{zhangq33,khedri,jaskolj}@mcmaster.ca
<http://www.cas.mcmaster.ca>

Abstract. Crosscutting concerns are pervasive in embedded software and ambient systems due to the stringent non-functional requirements imposed on them. Maintaining families of these systems to address issues with the crosscutting concerns, such as security concerns, is recognised to be tedious and costly. To tackle the above problem, we adapt the aspect-oriented paradigm to feature-modeling.

One of the most serious problems in aspect-oriented modeling is the potential of taking a valid model and spoiling its validity when weaving an aspect to it. We present a formal verification technique of aspectual composition in the context of feature-modeling that is based on product family algebra. We define a set of validity criteria for aspects with regard to their corresponding base specifications. The verification is done prior to the weaving of the aspects to their base specifications.

Keywords: Software product families, Aspect-oriented paradigm, Feature-modeling, Formal methods, Requirements verification.

1 Introduction

Product family engineering involves developing a variety of related products from core assets rather than developing them one-by-one independently. Feature-modeling techniques are widely used at the domain analysis stage to specify and manage the commonality and variability of product families in terms of features. In the literature, several feature-modeling techniques have been proposed using different notations and most of them are based on graphical notations [4]. In [8,9,10,11], Höfner et al. proposed a formal technique, *product family algebra*, to capture a set of different notations and terms found in current feature-modeling techniques. Product family algebra is notable for its capability to formally and concisely specify feature models. Other feature-modeling languages can be easily translated into that of product family algebra [2].

The development, maintenance, and evolution of complex and large feature models are among the main challenges faced by feature-modeling practitioners. Maintaining crosscutting concerns, such as security concerns, is recognised

* This research is supported by Natural Sciences and Engineering Research Council of Canada (NSERC) through the grant RGPIN227806-09.

in general software development as tedious and costly (e.g., [20]). These concerns are pervasive, especially in embedded software and ambient systems due to the various constraints imposed by the environment and the stringent non-functional requirements imposed on them. The difficulties of dealing with unpredictable changes to software requirements increase when a serious defect within a family is detected after putting the products of the family on the market. When the cause of the defect is spread to several parts of each of the products, the maintenance becomes quite tedious if the family is not properly modeled. The question then becomes how to quickly supersede the current feature model of a family by a new one to ensure that all the products that involve the identified configuration of features contributing to the problem get modified, replaced, or removed. We adopt the aspect-oriented paradigm to feature-modeling to tackle the above problems. We have proposed a language called AO-PFA [22] which contributes to a modular management of the commonality and variability of product families at the domain analysis stage of product family engineering. Aspect-oriented programming leads to systems with high modifiability, but at the same time the performance is hindered [14]. Also, the complexity of the programming languages makes the aspect weaving process very convoluted and prone to several aspectual compositional problems [18]. The language that we use at the feature-modeling level is indeed a lot simpler than a language used at the implementation level. That is why we conjecture that aspect-oriented techniques, despite their mixed results at the programming level, can be helpful at the feature-modeling level.

One of the most serious problems in aspect-oriented modeling is the potential of taking a model that is valid and spoiling its validity when weaving an aspect to it. At the feature-modeling level, the interference of an aspect with existing features, in a potentially undesired manner, can be caused either by the enforcement of the model by adding or amending existing features, products or families, or by relaxing the scope of a model by deleting features, products or families. Particularly, when an aspectual feature is imposed at a specific point in the feature model, it might alter the dependencies among features or, for instance, create cyclic definitions of composite features. The main contribution of this paper is a formal technique to verify aspectual composition in AO-PFA. While [22] presents the language of AO-PFA and highlights its relevance to feature modeling, the current work tackles the problem of aspectual composition within AO-PFA. We present a set of definitions and propositions enabling the verification of the validity of aspects with regard to their base specifications.

In Section 2 we briefly introduce the aspect-oriented specification language and other related background knowledge. In Section 3, we present the proposed formal approach to detect invalid aspects in the context of AO-PFA and highlight the usage of the proposed technique with several examples. In Section 4, we discuss and compare the proposed technique to those found in the literature, and give the highlights of our future work.

2 Background

2.1 Aspect-Orientation: Basic Concepts

To efficiently handle crosscutting concerns, the aspect-oriented paradigm (e.g., [12]) encapsulates crosscutting concerns by using aspects. In general, the whole system specification (or code) is obtained by composing aspects with the system's base specification (or code). This paradigm has been adapted to the whole software development life-cycle. From one software development stage to another, aspects are present in terms of fine or coarse granularity. For instance, at the pre-requirements stage, they are presented in terms of goals or features, while at the programming level, they are given in terms of program constructs. However, a terminology is widely and commonly used by the community of aspect-oriented software development. An aspect is composed of a *pointcut* and an *advice*. The advice describes what to realise by a crosscutting concern, while the pointcuts describe how to compose the concerns with other concerns. In base systems, points where aspects can be implicitly invoked are referred to as *join points*. The process to compose an aspect with a base system is called the *weaving process*. Roughly speaking, the weaving process introduces the advice to the base system at selected join points that are specified by the pointcut.

2.2 Product Family Algebra

Product family algebra [8,9,10,11] extends the mathematical notations of semiring to describe and manipulate product families. Precisely, a product family algebra is a commutative idempotent semiring $(S, +, \cdot, 0, 1)$. Each element of the semiring is a product family. Operator $+$ is interpreted as a choice between two product families and operator \cdot is interpreted as a mandatory composition of two product families. The element 0 represents the empty product family and the element 1 represents a product family consisting of only a pseudo-product which has no features.

Other concepts in product family modeling can be expressed mathematically in product family algebra. A product family a is the *subfamily* of a product family b , denoted by $a \leq b$, iff $a + b = b$. New product families can be derived from other existing families by adding features. A *refinement* relation is defined to capture such a relationship between two product families. Formally, a product family a is the refinement of a product family b , denoted by $a \sqsubseteq b$, iff $\exists(c \mid a \leq b \cdot c)$. Moreover, constraints are elicited in multi-view approaches when integrating different views. In the context of product families, a constraint can be informally described as “If a member of a product family has property P_1 , it must also (not) have property P_2 ” [10]. To capture such constraints in product family algebra, a *requirement* relation is defined.

Definition 1 (e.g., [10]). *For elements a, b, c, d and a product p in product family algebra, the requirement relation (\rightarrow) is defined in a family-induction style as follows:*

$$a \xrightarrow{p} b \stackrel{\text{def}}{\iff} p \sqsubseteq a \implies p \sqsubseteq b \quad \text{and} \quad a \xrightarrow{c+d} b \stackrel{\text{def}}{\iff} a \xrightarrow{c} b \wedge a \xrightarrow{d} b.$$

| | |
|--|---|
| 1. bf tim_con %time_controller | 20. Usr_Int = gra_tv · (1+web_bas) · (1+PDA) |
| 2. bf lum_sen %luminance_sensor | %User_Interface |
| 3. bf wea_sen %weather_sensor | 21. Commun = (cable+wireless+cable· wireless) |
| 4. bf gra_tv %graphical_tv | · (1+inter) · (1+mob_net) %Communication |
| 5. bf web_bas %web_based | 22. Lig_con = man_lig · (1+sma_lig) |
| 6. bf PDA %PDA | %Lighting_control |
| 7. bf cable %cable | 23. Daw_con = man_daw · (1+sma_daw) |
| 8. bf wireless %wireless | %Door_and_window_control |
| 9. bf inter %Internet | 24. HApp_con = (1+heat_sys) · (1+ster_sys) |
| 10. bf mob_net %mobile_phone_network | · (1+wat_intr) %Home_appliance_control |
| 11. bf man_lig %manual_lighting | 25. fir_det_flow = sma_lig · sma_daw · HApp_con |
| 12. bf sma_lig %smart_lighting | %fire_detection_flow |
| 13. bf man_daw %manual_door_and_window | 26. Home_gateway = Lig_con · Daw_con · HApp_con |
| 14. bf sma_daw %smart_door_and_window | · fir_det_flow %Home_gateway |
| 15. bf heat_sys %heating_system | 27. Home_Auto_PL = Commun · Usr_Int · Lig_dev |
| 16. bf ster_sys %stereo_system | · Daw_dev · Home_gateway |
| 17. bf wat_intr %water_intrusion | %Home_Automation_product_line |
| 18. Lig_dev = 1+lum_sen %Light_device | % Constraints |
| 19. Daw_dev = tim_con · (1+wea_sen) | 28. constraint(sma_lig, Home_Auto_PL, Lig_dev) |
| %Door_and_window_device | 29. constraint(sma_daw, Home_Auto_PL, Daw_dev) |
| | 30. constraint(web_bas, Home_Auto_PL, inter) |

Fig. 1. PFA Specification of a Home Automation Product Line

A requirement relationship $a \xrightarrow{f} b$, reading as “ a requires b within f ” indicates that for every product in f , if it is a refinement of a , then it must be a refinement of b .

2.3 Aspect-Oriented Product Family Algebra (AO-PFA)

The language AO-PFA [21,22] is an extension to the language of product family algebra. The motivation and benefits of AO-PFA can be found in [22]. Briefly, AO-PFA is used to articulate aspects (crosscutting concerns) in product family specifications. The maintainability of systems is enhanced by adopting the principle of separation of concerns. Different software system evolution demands can be captured as aspects where their specifications can be automatically composed/weaved with that of the system.

Specifications of product family algebra can be efficiently analysed and verified using a software tool Jory [2], which implements the automatic analysis of feature models based on binary decision diagrams. The specification language used by Jory is based on product family algebra and is called PFA. It involves three types of syntactic elements: basic feature declarations, labeled product families, and constraints. A *basic feature label* preceded by the keyword *bf* declares a basic feature. An equation with a *product family label* at the left-hand side and a product family algebra term at the right-hand side gives a labeled product family. A triple preceded by the keyword *constraint* represents a constraint, which specifies a requirement relation as introduced in Definition 11. Figure 1 gives an example of a PFA specification. Lines 1-17 are basic feature declarations, Lines 18-27 specify labeled product families, and Lines 28-30 correspond to constraints. For example, a basic feature, luminance sensor, is declared by *lum_sen* at Line 2, and a subfamily light device (represented by *Lig_dev*) that includes an optional feature *lum_sen* (represented by the product family term $(1 + lum_sen)$) is defined

at Line 18. Line 28 corresponds to the following constraint in product family algebra: $sma_lig \xrightarrow{\text{Home_Auto_PL}} Lig_dev$, which means that the feature sma_lig requires the feature Lig_dev in the family $Home_Auto_PL$.

In AO-PFA, the base is specified by a PFA specification, while a sequence of aspects is specified by an aspect specification. The aspect specification can handle multiple aspects weaved to one base specification, and the weaving order is naturally from top to bottom of the aspect specification. Moreover, each aspect should capture the basic concepts of pointcut and advice. Precisely, the syntax of an aspect is given as follows:

$$\begin{array}{l} \text{Aspect} \quad \text{aspectId} = \text{Advice}(jp) \\ \text{where } jp \in (\text{scope}, \text{expression}, \text{kind}) \end{array}$$

The label of an aspect is specified by $aspectId$. The body of the advice is represented by $\text{Advice}(jp)$. Since join points in PFA specifications are unified as product family terms, $\text{Advice}(jp)$ can be a general product family term; either a ground term or a term with a variable jp that represents the instance of a join point. Current aspect-oriented languages concur on the need of three basic attributes for a pointcut: the scope of join points, the kind of join points, and a boolean expression as a guard. We echo this consensus by representing the pointcut of an aspect in AO-PFA with a triple: $(\text{scope}, \text{expression}, \text{kind})$. The pointcut triple specifies rules for composing aspects and plays an important role when verifying aspectual composition. The expression of the pointcut triple works as a guard for the selected join points. The verification approach discussed in the next section primarily focuses on the analysis of the scope and the kind of the pointcut triple. The scope specifies boundaries of join points in the base specification. When the scope is the whole specification, we write “base” in the field of scope. Two basic types of specific scopes, within and hierarchy , are designed to respectively capture join points within the lexical structures or hierarchical structures of the specified product families. Moreover, $\text{protect}(\text{scope})$ is used to specify that eligible join points are excluded from the scope, and two scopes can be combined with “:” and “;” to respectively indicate the union and intersection of scopes. The kind selects join points based on the exact positions and forms of the product family algebra terms. There are seven types of pointcut kinds. Pointcut kinds of declaration and inclusion refer to join points at basic feature declarations in PFA specifications. Pointcut kinds of creation , $\text{component_creation}$, component and $\text{equivalent_component}$ refer to join points at labeled product families in PFA specifications. Join points within constraints in PFA specifications are captured by a pointcut kind of $\text{constraint}[\text{position_list}]$. Obviously, each join point can only belong to one pointcut kind. Due to space limitations, we explain the detailed semantics of different types of pointcuts only when necessary. We refer the reader to [21,22] for details and examples showing the usage of those pointcuts.

A home automation product line adapted from [16] is used as a case study to illustrate the application of AO-PFA in [21]. A home automation system includes control devices, communication networks, user interfaces, and a home gateway.

Different types of devices, network standards, and user interfaces can be selected for different products. A home gateway offers different services for overall system management. In this paper, we only use parts of the case study to show the usage of the proposed verification technique for aspectual composition. Figure 1 is a PFA specification of the home automation product line. The PFA specification is used as the base specification and is composed with different aspects in Section 3. We will show the problems caused by composing those aspects, and associate them to the presented validity criteria.

3 Verification of Aspects at the Feature-Modeling Level

Adopting aspect-orientation at early stages facilitates a systematic aspect-oriented development of product families throughout the whole life-cycle, which is especially important for model-driven software development [5]. A major type of interference caused by aspectual composition is derived from early development stages and should be handled earlier [18]. In the context of product families, the safe feature composition problem has been widely studied (e.g., [13]), and is quite challenging to tackle. The compositions of features can cause conflicts either by the behavior of features, the incompleteness of feature models, or both [13]. Instead of detecting all invalid feature compositions at the detailed level of features, adapting the aspect-oriented paradigm to feature-modeling helps to detect some invalid feature compositions at the abstract level of features by verifying the aspectual composition, which comes to the main contribution of this paper. With such an early detection, we can make necessary tradeoffs as early as possible at the feature-modeling stage, and provide valuable knowledge for the following design and implementation stages for product families.

In this section, we address the verification of aspectual composition in AO-PFA specifications. In general, we formalise the validity criteria for PFA specifications and analyse the impacts of aspects on PFA specifications.

3.1 Validity Criteria for PFA Specifications

We first establish our mathematical settings to identify what criteria need to be satisfied for a valid PFA specification representing a product family feature model. In PFA specifications, the most basic constructs are those labels that either represent features or product families. Therefore, we abstract validity criteria of PFA specifications at the finest granularity with regard to those labels.

Construction 1. *Given a PFA specification S , let D_S be the multi-set of labels that are present in S at basic feature declarations or at the left-hand sides of labeled product families. We call D_S the defining label multi-set associated with the specification S .*

Definition 2. *[Definition-valid specification] We say that a PFA specification S is definition-valid iff $\forall(v \mid v \in D_S : \text{NumOccur}(v) = 1)$ where D_S is the defining label multi-set of S , and $\text{NumOccur}(v)$ denotes the number of occurrence of v in D_S .*

Construction 2. *Given a PFA specification S , let R_S be the set of labels that are present in S at the constraints or at the right-hand sides of labeled product families. We call R_S the referencing label set associated with the specification S .*

Definition 3. *[Reference-valid specification] We say a specification S is reference-valid iff $R_S \subseteq D_S$, where R_S is the referencing label set of S and D_S is the defining label multi-set of S .*

Intuitively, Definition 2 indicates that a specification is definition-valid iff all the elements in D_S are unambiguously defined labels. If the multi-set D_S actually forms a set, we call D_S the defining label set associated with a specification S . Definition 3 indicates that a specification is reference-valid iff all the elements in R_S are not references to undefined labels. As an example, according to Constructions 1 and 2, the defining label multi-set and the referencing label set of the specification *home_automation* (Figure 1) are identical (i.e., $D_{\text{home_automation}} = R_{\text{home_automation}} = \{\text{tim_con, lum_sen, wea_sen, gra_tv, web_bas, PDA, cable, wireless, inter, mob_net, man_lig, sma_lig, man_daw, ster_sys, sma_daw, heat_sys, wat_intr, fir_det_flow, Lig_dev, Daw_dev, Ustr_Int, Commun, Lig_con, HApp_con, Home_gateway, Home_Auto_PL, Daw_con}\}$). Based on Definitions 2 and 3, we claim that the specification *home_automation* is definition-valid and reference-valid.

Construction 3. *Given a PFA specification S , let D_S be its corresponding defining label set and $G_S = (V, E)$ be a digraph. The set of vertices $V \subseteq D_S$, and a tuple (u, v) is in E iff u occurs in a product family term T such that the equation $v = T$ is a labeled product family specified in S . We call G_S the label dependency digraph associated with the specification S .*

Let $G_S = (V, E)$ be a label dependency digraph associated with a PFA specification S . For $u, v \in V$, we say that u defines v iff $\exists(n \mid n \geq 1 : (u, v)\text{-path} \in E^n)$. Consequently, we say u and v are *mutually defined* labels, denoted by $\text{mutdef}(u, v)$, iff $\exists(m, n \mid m, n \geq 1 : (u, v)\text{-path} \in E^m \wedge (v, u)\text{-path} \in E^n)$. In particular, if u and v are identical and $m = n = 1$, we say u (or v) is *self-defined*.

Definition 4. *[Dependency-valid specification] We say that a PFA specification S is dependency-valid iff $\forall(u, v \mid u, v \in V : \neg \text{mutdef}(u, v))$ where V and $\text{mutdef}(u, v)$ are defined according to $G_S = (V, E)$ that is the label dependency digraph associated with the specification S .*

Intuitively, Definition 4 indicates that a valid PFA specification does not have any *mutually defined* or *self-defined* labels.

Lemma 1. *Let $G_S = (V, E)$ be a label dependency digraph of a PFA specification S . Two labels u and v are mutually defined iff there is a cycle including u and v in G_S . In particular, a label u is self-defined iff there is a loop through u in G_S .*

The detailed proof of Lemma 1 can be found in [21]. Straightforwardly, the digraph G_S associated with a dependency-valid specification S should be cycle-free and loop-free. For example, the dependency digraph of the specification

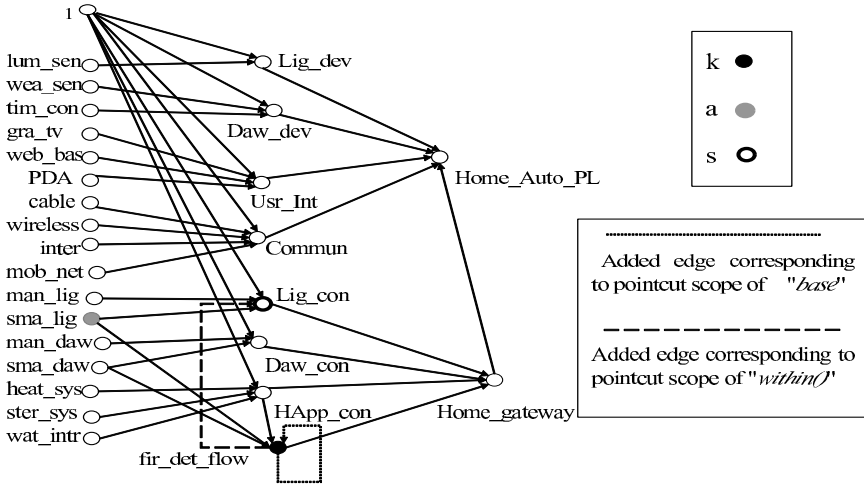


Fig. 2. Dependency digraph corresponding to the home automation product line

home_automation (Figure 1) can be constructed as given in Figure 2. The digraph is loop and cycle free, which indicates that the corresponding specification is dependency-valid.

3.2 Validity Criteria of Aspects in AO-PFA

In AO-PFA, aspects are composed with base specifications at the granularity of product family terms. The obligation of aspectual composition is that the weaved specifications resulting from composing aspects and base specifications should remain valid. We first consider the following examples to illustrate that weaving an aspect to a base specification may cause definition-invalid, reference-invalid and dependency-invalid specifications.

Aspects Causing definition-invalid Specifications: We consider the following two aspects that are developed independently. The aspect denoted as *case1a_aspect* intends to deploy a fingerprint reader device as an optional feature in the sub-family *Daw_dev*, while the aspect denoted as *case1b_aspect* intends to deploy the fingerprint reader device as a mandatory feature. We denote the fingerprint reader by *fgr*. The two aspects can be respectively specified in AO-PFA as follows:

| | |
|---|---|
| <p>case1a_aspect: Aspect $jp_new = jp \cdot (1 + fgr)$ where $jp \in (base, true, creation(Daw_dev))$</p> | <p>case1b_aspect: Aspect $jp_new = jp \cdot fgr$ where $jp \in (base, true, creation(Daw_dev))$</p> |
|---|---|

The *creation* pointcut refers to join points at the exact definitions of labeled product families. Let the base specification associated with the above two aspects be the specification *home_automation* (Figure 1). Both aspects will capture the left-hand side of the labeled product family *Daw_dev* at Line 19 in Figure 1 and

introduce a new labeled product family *Daw_dev_new*. In other words, weaving these two aspects to the same base specification can result in a definition-invalid PFA specification.

Aspects Causing reference-invalid Specifications: We consider another aspect (denoted as *case2_aspect*) that extends the fingerprinter reader *fgr*, with a new feature *password_identify*. The aspect is specified as follows:

case2_aspect:
 Aspect $jp = jp \cdot password_identify$
 where $jp \in (base, true, inclusion(fgr))$

The *inclusion* pointcut refers to join points at the reference of the feature *fgr*. Assume that we want to weave both aspects *case1a_aspect* and *case2_aspect* to the base specification *home_automation* (Figure 11). If we weave *case2_aspect* first, there is actually no modification to the base specification after the weaving process as the feature *fgr* does not appear in the base specification. The feature *fgr* then appears in the new specification after weaving *case1a_aspect*. However, *case2_aspect* does not take effect, which may not correspond to our expectation. Actually, weaving the first aspect leads to a reference-invalid specification. It is more reasonable to weave *case1a_aspect* before *case2_aspect*.

Aspects Causing dependency-invalid Specifications: We consider composing the aspect specified below (denoted as *case3_aspect*) with the base specification *home_automation* (Figure 12):

case3_aspect:
 Aspect $jp = jp \cdot fir_det_flow$
 where $jp \in (base, true, inclusion(sma_lig))$

The dependency digraph of the specification *home_automation* is given in Figure 12. The dotted edge illustrates the new edge introduced by *case3_aspect*, which introduces a loop to the dependency digraph. The example shows that weaving an aspect might lead to a dependency-invalid specification. Alternatively, assume that we change the scope of the above pointcut to be *within(Lig_con)* instead of *base*, the dashed edge introduced by the modified aspect will not cause loops or cycles in the dependency digraph.

Formalisation of Aspectual Composition. Instead of checking the validity of specifications after weaving the aspects, our approach intends to detect the above validity problems before the weaving process. Besides, it is necessary for us to formalise such verification to enable automation. With regard to the validity definitions of PFA specifications, composing an aspect to a base specification may change the defining label multi-set, the referencing label set, and the label dependency digraph of the original specification. Precisely, the effects of weaving an aspect can be abstracted with the following construction.

Table 1. The effect of pointcut kinds on D_A and R_A

| kind of pointcut | D_A contains | R_A contains |
|--|---|---|
| <i>inclusion</i> | all newly introduced labels specified by $Advice(jp)$ | all labels specified by $Advice(jp)$ ¹ |
| <i>component</i> | | |
| <i>equivalent_component</i> | | |
| <i>declaration</i> | all newly introduced labels specified by $Advice(jp)$ and all labels in $aspectId$ ² | |
| <i>creation</i> | | |
| <i>component_creation</i> | empty set | |
| <i>constraint[list]</i> | | |
| ¹ including the instance of jp in the case where jp appears in $Advice(jp)$ | | |
| ² including the instance of jp in the case where jp appears in $aspectId$ | | |

Construction 4. Let S' be the PFA specification obtained by weaving an aspect A to a valid PFA specification S . The defining label sets, referencing label sets and dependency digraphs of S and S' can be constructed according to Constructions 7-8, respectively. To discuss the difference between S' and S , we denote D_A , R_A , E_add_A and E_del_A associated with the aspect A as follows:

- Let D_A be a set of labels introduced by A which will be present at basic feature declarations or left-hand sides of labeled product families in S' . As every element $v \in D_A$ is a defining label, the defining label multi-set of S' is $D_{S'} = D_S \sqcup D_A$, where \sqcup denotes the multi-set union.
- Let R_A be a set of labels introduced by A which will be present at constraints or right-hand sides of labeled product families in S' . As every element $v \in R_A$ is a referencing label, the referencing label set of S' is $R_{S'} = R_S \cup R_A$.
- Let E_add_A be a set of tuples (u, v) such that u is a label that will be introduced by aspect A at the right-hand side of a labeled product family in S' and v is the label present at the left-hand side of the labeled product family. Let E_del_A be a set of tuples (u, v) such that the label u will be removed by A from the right-hand side of a labeled product family in S , and v is the label present at the left-hand side of the labeled product family. As E_add_A and E_del_A correspond to edge additions and deletions in G_S , the dependency digraph of S' is $G_{S'} = (V, (E_S \cup E_add_A) - E_del_A)$ where $V \subseteq D_{S'}$.

Consequently, an aspect A is invalid w.r.t. a specification S iff the weaved specification S' is definition-invalid, reference-invalid or dependency-invalid according to Definitions 2-4, respectively.

Detection of Definition-Invalid and Reference-Invalid Aspects. Since the pointcut kind decides the position of join points, we can directly construct D_A and R_A with regard to different pointcut kinds as given in Table 1. Pointcut kinds of *declaration*, *creation*, and *component_creation* introduce new specifications where the specified product families are defined. Pointcut kinds of *inclusion*, *component*, *equivalent_component*, and *constraint[position_list]* introduce new specifications where the specified product families are referenced. With accordance to Construction 4, the formal definitions for definition-valid aspect and reference-valid aspects can be given by Definition 5 and 6, respectively.

Definition 5. [*Definition-valid aspect*] We say that an aspect A is definition-valid with regard to a specification S iff $D_S \cap D_A = \emptyset$.

Considering the example given earlier, we assume that, without loss of generality, $case1a_aspect$ is weaved before $case1b_aspect$. We construct the defining label set of the aspect according to the row corresponding to *creation* in Table 1 and obtain $D_{case1a_aspect} = D_{case1b_aspect} = \{fgr, Daw_dev_new\}$. According to Definition 5, since the intersection of $D_{home_automation}$ and D_{case1a_aspect} is empty, the aspect $case1a_aspect$ is definition-valid w.r.t. its base specification. Let $home_automation_one$ be the specification after weaving $case1a_aspect$. According to Construction 4, we have $D_{home_automation_one} = D_{home_automation} \sqcup D_{case1a_aspect}$. As the intersection of $D_{home_automation_one}$ and D_{case1b_aspect} is nonempty, the aspect $case1b_aspect$ is definition-invalid w.r.t. its base specification $home_automation_one$. Actually, the problem in the example is caused by the ambiguous definitions of a subfamily from two independently developed aspects. The proposed verification leads to formally detecting such conflict at the feature-modeling level and enabling tradeoffs at earlier development stages.

Definition 6. [*Reference-valid aspect*] We say that an aspect A is reference-valid with regard to a PFA specification S iff $(R_S \cup R_A) \subseteq (D_S \cup D_A)$.

According to the row of *inclusion* in Table 1, we construct the defining label set and referencing label set corresponding to the example of $case2_aspect$. We obtain $D_{case2_aspect} = \{password_identify\}$, $R_{case2_aspect} = \{fgr, password_identify\}$. Based on Definition 6, $case2_aspect$ is reference-invalid w.r.t. the specification $home_automation$. On the other hand, suppose we weave $case1a_aspect$ before $case2_aspect$. Let $home_automation_one$ be the base specification associated with $case2_aspect$. According to Table 1 and Construction 4, $R_{home_automation_one} = R_{home_automation} \cup \{fgr, Daw_dev\}$. Consequently, $case2_aspect$ becomes reference-valid w.r.t. the specification $home_automation_one$. In this example, the violation is indeed caused by introducing a feature reference to a undefined feature. Detecting such invalid aspectual composition at the feature-modeling level can help us to decide the correct composition order of features at an early stage.

Detection of Dependency-Invalid Aspects. Unlike detecting violations of definition-validity and reference-validity, detecting the violation of dependency-validity of an aspect is not straightforward. The conventional way to detect such dependency-invalid aspects is to construct the dependency digraph from the weaved specifications and detect cycles and loops in the digraph. However, such an approach would be time-consuming and the weaving process would be unnecessary if the aspect is invalid w.r.t. a base specification. In order to detect such dependency-invalid aspects prior to the weaving process, we formalise the complex relations between the given aspect specification and the dependency digraph of a base specification to verify if the aspectual composition should be allowed.

With regard to Construction 4, instead of examining all edges in E_{add_A} and E_{del_A} , we only consider the edges that may introduce loops or cycles in G_S .

We use the following construction to identify vertices on the dependency digraph in a base specification, which are directly affected by weaving an aspect.

Construction 5. *Let G_S be a dependency digraph associated with a valid specification S . We rename or create vertices in G_S with regard to an aspect A as follows:*

- We denote a vertex in G_S corresponding to the term specified by the pointcut kind of A by k . When there is no such vertex, a new vertex is created and named k .
- We denote a vertex in G_S corresponding to the label specified by the pointcut scope of A by s .

Proposition 1. *[Pointcut Kind Condition] An aspect A is dependency-valid w.r.t. a valid PFA specification S if the kind of pointcut of A is “constraint[list]”.*

Proof. When the kind of pointcut is *constraint[list]*, the aspect A only affects the product families within the constraints. According to Construction 4, it indicates that $D_A = E_add_A = E_del_A = \emptyset$. Consequently, we accomplish the proof by proving $(D_A = E_add_A = E_del_A = \emptyset \implies A \text{ is dependency-valid w.r.t. } S)$. We refer the reader to [21] for the detailed proof. \square

We define a function $Walk: V \times V \rightarrow ordered-list(V)$ over a digraph. $Walk(u, v)$ returns the list of all vertices along a walk from u to v . A valid label dependency digraph is a typical digraph that can have a topological ordering. A walk between two vertices is indeed a path. Therefore, in the label dependency digraph, the vertex list $Walk(u, v)$ is sufficient to identify a path from u to v . Particularly, if $Walk(u, v)$ is empty, it indicates that there is no path from u to v .

Proposition 2. *[Non-cycle Condition] Let S be a valid PFA specification and A be an aspect that does not satisfy the pointcut kind condition (Proposition 1). Construct the dependency digraph G_S according to Construction 3 and denote or create the vertex k in G_S according to Construction 5. Then A is dependency-valid w.r.t. S if $\forall(x \mid x \in D_S \cap R_A : Walk(k, x) = \emptyset)$.*

The proof is quite long and we refer reader to [21] for the complete detailed proof. From the proof, we have the equation:

$$A \text{ is dependency-invalid w.r.t. } S \iff \exists(v \mid v \in D_S \cap R_A : v \in \mathbb{JP}_{\text{def}} \vee \exists(u \mid u \in \mathbb{JP}_{\text{def}} : \exists(m \mid m \geq 1 : (u, v)\text{-path} \in (E_S)^m)) \quad (1),$$

where \mathbb{JP}_{def} is the set of family labels of labeled product families where join points are present at their right-hand sides.

We say that an aspect is potentially dependency-invalid if it does not satisfy both the *pointcut kind condition* (Proposition 1) and the *non-cycle condition* (Proposition 2). If a potentially invalid aspect is detected, we further verify whether it is actually invalid with regard to its base specification according to its pointcut scope.

Proposition 3. *[Dependency-invalid aspect] Let S be a valid PFA specification and A be a potentially dependency-invalid aspect. Denote the vertex that invalidates the condition of Proposition 2 by a . Vertices k and s are denoted or created in G_S as prescribed in Construction 5. Let $Dep_invalid(ts)$ be the following predicate, where ts represents the type of the pointcut scope.*

$$Dep_invalid(ts) \stackrel{\text{def}}{\Leftrightarrow} \begin{cases} \text{true} & \text{if } ts \text{ is base} \\ s \in Walk(k, a) \wedge s \neq k & \text{if } ts \text{ is within} \\ s \in Walk(k, a) \wedge s \neq k \wedge s \neq a & \text{if } ts \text{ is hierarchy} \\ \neg Dep_invalid(ts') & \text{if } ts \text{ is protect}(ts') \\ Dep_invalid(ts_1) \vee Dep_invalid(ts_2) & \text{if } ts \text{ is } (ts_1 : ts_2) \\ Dep_invalid(ts_1) \wedge Dep_invalid(ts_2) & \text{if } ts \text{ is } (ts_1 ; ts_2) \end{cases}$$

Provided the set of join points is nonempty, the aspect A is dependency-invalid w.r.t. S if $Dep_invalid(ts)$.

The proof of the above proposition is based on Lemmas 2, 3, and 4 given in the Appendix. The complete detailed proof can be found in [21].

We revisit the previously introduced example corresponding to Figure 2. In the digraph, we represent vertices that are in both the referencing label set of the aspect and the defining label set of the base specification by grey vertices. The vertices k and s are represented by the black vertex and bold circle vertex, respectively. Based on Proposition 2 and the first item in Proposition 3, the aspect *case3_aspect* is dependency-invalid w.r.t. its base specification; there is a path from the vertex of *sma_lig* to the vertex of *fir_det_flow*. On the other hand, based on the second item in Proposition 3, the modified aspect with a bounded scope is dependency-valid w.r.t. its base specification. The above verification results confirm the results we obtained by detecting loops and cycles. However, we do not need to construct the new edges introduced by aspects. It is indicated that the aspect in this example should be created by specifying scopes, i.e., the features should be composed to the base family within a certain scope.

4 Related Work, Discussion, and Conclusion

In our work, the aspect model composition is applied to feature models. A related work of the composition of feature models can be found in [1]. Unlike AO-PFA, Acher et al. [1] mainly focus on the insert and merge operators of feature models. Their work considers the composition operators from the perspective of model integration, whereas our work discusses the issue from the perspective of composition mechanisms for different concerns. We refer the reader to [9] for the integration of several views of a feature model using product family algebra.

Currently, many approaches have been proposed to adapt the aspect-oriented paradigm to product family engineering at different abstract levels and different development stages. For example, [19] discusses the usage of aspect-orientation for variability implementation, management, and tracing throughout the whole lifecycle of product family engineering. AML (Aspectual Mixin

Layers) [3] introduces the aspect-oriented paradigm into feature-orientation programming. Xweave [6] applies aspects to design models of product families. Hi-Transform [15] presents an approach to extend the aspect-oriented paradigm to the model transformation. To the best of our knowledge, no other works have attempted the introduction of the aspect-oriented paradigm at the formal feature-modeling level.

In the literature of the aspect-oriented paradigm at the early requirement and design stages, most approaches are informal. Those informal approaches are easy to understand and are suited for user validation, but the verification in those approaches is only accomplished by informally “walking through” the artifacts [5]. By constructing our technique upon programming-like formal specifications, our verification technique is inspired by a static code analysis approach described in [17] that characterises the direct and indirect interactions of aspects with base systems. Although analogised from AspectJ [12], the notations used in AO-PFA are simplified and unified with the help of product family algebra.

Our paper is also related to safe feature compositions. A collection of existing approaches for safe feature compositions are discussed at the detailed level of features (e.g., [7]). The approach in [13] formalises the existence-identifiers and non existence-identifiers for composing features using proposition logic, and SAT solvers are used to verify the correctness of feature composition. In our approach, we formalise feature models based on product family algebra, and we detect unsafe compositions at the abstract level of features by verifying aspectual composition in AO-PFA. Precisely, to check Definitions 5 and 6 for definition-validity and reference-validity of aspects, it is straightforward to see that the complexities are both $O(|V|)$, where $|V|$ is the cardinality of the set of vertices of the dependency digraph of the base specification. To check Propositions 1-3 for the dependency-validity of the aspect, by using graph algorithms (e.g., depth-first search) the complexity is $O(|V| + |E|)$, where E is the set of edges and $|E|$ is less or equal to $|V|^2$.

Our short term future work focuses on automating the aspectual verification as presented in this paper. Moreover, even if the aspects given in terms of features (i.e., as black-boxes) do not invalidate the feature model according to the definitions and propositions given in this paper, features might still interact in an undesirable way at the detailed level. We intend to tackle this issue as future work based on the work presented in [11].

References

1. Acher, M., Collet, P., Lahire, P., France, R.: Composing Feature Models. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 62–81. Springer, Heidelberg (2010)
2. Alturki, F., Khedri, R.: A Tool for Formal Feature Modeling Based on BDDs and Product Families Algebra. In: 13th Workshop on Requirement Engineering (2010)
3. Apel, S., Leich, T., Saake, G.: Aspectual Mixin Layers: Aspects and Features in Concert. In: Proc. of Intl. Conf. on Software Engineering, pp. 122–131. ACM, N.Y. (2006)

4. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated Analysis of Feature Models 20 Years Later: A Literature Review. *J. Information Systems* 35(6), 615–636 (2010)
5. Chitchyan, R., Rashid, A., Sawyer, P., et al.: Survey of Analysis and Design Approach. Tech. rep., AOSD-Europe (2005)
6. Groher, I., Voelter, M.: Xweave: Models and aspects in Concert. In: Proc. of the 10th Workshop on Aspect-Oriented Modelling, pp. 35–40. ACM, N.Y. (2007)
7. Hannousse, A., Douence, R., Ardourel, G.: Static Analysis of Aspect Interaction and Composition in Component Models. In: 10th Intl. Conf. on Generative Programming and Component Engineering, pp. 43–52. ACM, N.Y. (2011)
8. Höfner, P., Khedri, R., Möller, B.: Feature Algebra. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 300–315. Springer, Heidelberg (2006)
9. Höfner, P., Khedri, R., Möller, B.: Algebraic View Reconciliation. In: The 6th IEEE Intl. Conf. on Software Engineering and Formal Methods, pp. 85–94. IEEE CS, Wash. (2008)
10. Höfner, P., Khedri, R., Möller, B.: An Algebra of Product Families. *Software and Systems Modeling* 10(2), 161–182 (2011)
11. Höfner, P., Khedri, R., Möller, B.: Supplementing Product Families with Behaviour. *Intl. J. of Software and Informatics* 5(1–2), 245–266 (2011)
12. Kiczales, G., Lamping, J., Mendhekar, A., et al.: Aspect-Oriented Programming. In: Aksit, M., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
13. Kuhlemann, M., Batory, D., Kästner, C.: Safe Composition of Non-Monotonic Features. In: Proc. of 8th Intl. Conf. on Generative Programming and Component Engineering, pp. 177–186. ACM, N.Y. (2009)
14. Kuusela, J., Tuominen, H.: Aspect-Oriented Approach to Operating System Development Empirical Study. In: Elleithy, K. (ed.) Advanced Techniques in Computing Sciences and Software Engineering, pp. 233–238. Springer, Netherlands (2010)
15. Oldevik, J., Haugen, Ø.: Higher-Order Transformations for Product Lines. In: Proc. of the 11th Intl. Software Product Line Conf. IEEE CS, Wash. (2007)
16. Pohl, K.: *Software Product Line Engineering: Foundations, Principles, and Techniques*, ch. 3. Springer, N.Y. (2005)
17. Rinard, M., Salcianu, A., Bugrara, S.: A Classification System and Analysis for Aspect-Oriented Programs. *SIGSOFT Softw. Eng. Notes* 29(6), 147–158 (2004)
18. Sanen, F., Chitchyan, R., Bergmans, L., Fabry, F., Sudholt, M., Mehner, K.: Aspects, Dependencies and Interactions. In: Cebulla, M. (ed.) ECOOP-WS 2007. LNCS, vol. 4906, pp. 75–90. Springer, Heidelberg (2008)
19. Voelter, M., Groher, I.: Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In: Proc. of the 11th Intl. Software Product Line Conf., pp. 233–242. IEEE CS, Wash. (2007)
20. Xu, B., Yang, M., Liang, H., et al.: Maximizing Customer Satisfaction in Maintenance of Software Product Family. In: Canadian Conf. on Electrical and Computer Engineering, pp. 1320–1323 (2005)
21. Zhang, Q., Khedri, R., Jaskolka, J.: An Aspect-Oriented Language Based on Product Family Algebra: Aspects Specification and Verification. Tech. rep., McMaster University (2011)
22. Zhang, Q., Khedri, R., Jaskolka, J.: An Aspect-Oriented Language for Product Family Specification. In: The 3rd Intl. Conf. on Ambient Systems, Networks and Technologies, p. 10 (2012)

Appendix: Additional Results Needed for the Proof of Proposition 3

In this Appendix, we give the most important lemmas required for the proof of Proposition 3. We also give the highlights of their proofs. The reader can find all of the detailed proofs in [21].

Lemma 2. *Let S be a valid PFA specification and A be a potentially dependency-invalid aspect. When the scope of the pointcut is “base”, A is always dependency-invalid w.r.t. S . When the scope of the pointcut is “protect(base)”, A is always dependency-valid w.r.t. S .*

Proof. We use Equation (1) and substitute the definition of JP_{def} . When the type of the pointcut scope is *base*, join points are where k is present. Therefore, $\text{JP}_{\text{def}} = N^+(k)$, where $N^+(k)$ denote the set of all successors of the vertex k . When the type of the pointcut scope is *protect(base)*, the set JP_{def} is empty. In the proof, we use the definition of $\text{Walk}(u, v)$, path concatenation, the one-point rule, range split, empty range and \exists -false body. \square

Lemma 3. *Let S be a valid PFA specification and A be a potentially dependency-invalid aspect. Construct the dependency digraph G_S according Construction 3 and denote or create the vertex k in G_S according Construction 5.*

- *When the type of the pointcut scope is “within”, A is dependency-invalid w.r.t. S iff $\exists(v \mid v \in D_S \cap R_A : s \in \text{Walk}(k, v) \wedge s \neq k)$.*
- *When the type of the pointcut scope is “protect(within)”, A is dependency-invalid w.r.t. S iff $\exists(v \mid v \in D_S \cap R_A : s \notin \text{Walk}(k, v) \vee s = k)$.*
- *When the type of the pointcut scope is “hierarchy”, A is dependency-invalid w.r.t. S iff $\exists(v \mid v \in D_S \cap R_A : s \in \text{Walk}(k, v) \wedge s \neq k \wedge s \neq v)$.*
- *When the type of the pointcut scope is “protect(hierarchy)”, A is dependency-invalid w.r.t. S iff $\exists(v \mid v \in D_S \cap R_A : s \notin \text{Walk}(k, v) \vee s = k \vee s = v)$.*

Proof. Equation (1) and Lemma 2 are used in the proof. When the type of the pointcut scope is *within*, join points are bound to a labeled product family whose label is s . Therefore, we have $\text{JP}_{\text{def}} = \{s\}$. When the type of the pointcut scope is *hierarchy*, join points are bound to a labeled product family where s is present at the right-hand side. Therefore, $\text{JP}_{\text{def}} = N^+(s)$. Besides, there should be a path from k to s . On the other hand, when the type of the scope is *protect(within)*, labels in JP_{def} should not include s if there is a path from k to s . When the type of the scope is *protect(hierarchy)*, labels in JP_{def} should not include successors of s if there is a path from k to s . Otherwise, the set JP_{def} is identical to the one specified by a pointcut with scope *base*. \square

Lemma 4. *Let S be a valid PFA specification and A be a potentially dependency-invalid aspect. When the scope of pointcut is the union of two scopes of types ts_1 and ts_2 , A is dependency-invalid when A is dependency-invalid w.r.t. at least one of the scopes. When the scope of pointcut of A is the intersection of two scopes of types ts_1 and ts_2 , A is dependency-invalid when A is both dependency-invalid w.r.t. the two scopes.*

Proof. Let the set of join points selected by a pointcut scope of type ts_1 be JP_{def}^1 and the set of join points selected by a pointcut scope of type ts_2 be JP_{def}^2 . When the type of the pointcut scope is $(ts_1 : ts_2)$, then the set $\text{JP}_{\text{def}} = \text{JP}_{\text{def}}^1 \cup \text{JP}_{\text{def}}^2$. When the type of the pointcut scope is $(ts_1 ; ts_2)$, then the set $\text{JP}_{\text{def}} = \text{JP}_{\text{def}}^1 \cap \text{JP}_{\text{def}}^2$. Provided that $\text{JP}_{\text{def}} \neq \emptyset$, we substitute the definition of JP_{def} in Equation (1) and use Lemmas 2 and 3 to accomplish the whole proof. \square

A Denotational Model for Instantaneous Signal Calculus

Yongxin Zhao, Longfei Zhu, Huibiao Zhu, and Jifeng He*

Shanghai Key Laboratory of Trustworthy Computing,
Software Engineering Institute,
East China Normal University, Shanghai, China
{yxxzhao, lfzhu, hbzhu, jifeng}@sei.ecnu.edu.cn

Abstract. In this paper we explore an observation-oriented denotational semantics for instantaneous signal calculus which contains all conceptually instantaneous reactions of signal calculus for event-based synchronous languages. The healthiness conditions are studied for especially dealing with the emission of signals. Every instantaneous reaction can be identified as denoting a healthiness function over the set of events which describe the state of the system and its environment. The normal form, surprisingly, has the comparatively elegant and straightforward denotational semantic definition. Furthermore, a set of algebraic laws concerning the distinct features for instantaneous signal calculus is investigated. All algebraic laws can be established in the framework of our semantic model, i.e., if the equality of two differently written instantaneous reactions is algebraically provable, the two reactions are also equivalent with respect to the denotational semantics.

1 Introduction

With the continual development of computer science and IT industry, real-time systems [7,11] are designed to cater to many applications ranging from simple home appliances and laboratory instruments to complex control systems for chemical and nuclear plants, flight guidance of aircrafts, ballistic missiles and Cyber-Physical systems (CPS) [8], etc. In essential, the correctness of real-time systems not only depends on the result of logical computing, but also the result taken at the right time. Hence, compared to traditional non-real-time software systems, real-time systems have relatively rigid requirements and specifications; responses to external events should be within strict bounded time limits; the external environment tends to be greatly terrible and highly nondeterministic. All these characteristics bring serious challenges for the design of real-time systems, thus the research on the theory of real-time systems facilitates the description, analysis, design, implementation and verification of real-time systems.

Formal methods with rigorously mathematical description and verification techniques are considered as approaches to ensure real-time system requirements always correct and satisfied in the full development process from logical specification to physical implementation. The software engineers focus on decomposition of the computational activities into periodic tasks and set task priorities, and then select the scheduling policy

* Corresponding author.

to meet the desired hard real-time constraints. To separate the design task of software engineers from the implementation task, the formal languages have to provide an intermediate level of abstraction, which makes possible to leave the physical realisation job to the compiler.

Inspired by the Esterel language [112], we propose a signal calculus which is an event-based synchronous language for reasoning about embedded real-time systems. Our calculus adopts the so-called synchronous hypothesis [10,14], i.e., instantaneous reaction to signals and immediate propagation of signals in each time-instant. In [16], the algebraic semantics of the instantaneous signal calculus which contains all conceptually instantaneous reactions has been completely explored. A set of algebraic axioms is provided to describe the characteristic properties of the primitives and the combinators. The great advantage of algebra is that it uses equational (and inequational) reasoning, with a single conceptual framework and notation throughout [493]. Further, the corresponding algebraic normal form is provided and every instantaneous reaction, however deeply structured, can be reduced into the normal form by a series of algebraic manipulation. A term rewriting system can check the correctness of the reductions, and it can also carry out the compilation task [6]. Consequently, that two differently written instantaneous reactions happen to mean the same thing can be proved from the equation of their algebraic presentations.

In this paper, our intention is to explore an observation-oriented denotational semantics for instantaneous signal calculus. The healthiness conditions are studied for especially dealing with the emission of signals. Every instantaneous reaction can be identified as denoting a healthiness function over the set of events which record the observation of signal state. The normal form, surprisingly, has the comparatively elegant and straightforward denotational semantic definition. Thus the normal form indeed exposes the internal dependence of reactions in parallel branches and describes the intrinsic behaviour of instantaneous reactions. More importantly, with the help of the theory of normal form, we fortunately obtain another effective approach to the semantic definitions of instantaneous reactions. From the practical point of view, semantic definitions are achieved by term rewriting rather than direct semantic computing, and the algebraic construction is also amenable to efficient implementation by computer [12].

Furthermore, all algebraic laws concerning the distinct features for instantaneous signal calculus can be established in the framework of our semantic model, i.e., if the equality of two differently written instantaneous reactions is algebraically provable, the two reactions are also equivalent with respect to the denotational semantics. By demonstrating the validity of the algebraic laws, in fact, we prove the soundness of algebraic method to semantics in [16].

The remainder of the paper is organized as follows. Section 2 gives a brief introduction to the instantaneous signal calculus. Section 3 is devoted to explore the observation-oriented denotational semantics for primitives and combinators of the instantaneous signal calculus. The healthiness conditions of instantaneous reactions are also provided to construction the semantic domain. A set of algebraic laws of the instantaneous signal calculus is investigated in Section 4. we demonstrate that all algebraic laws can be established in the framework of our semantic model. Section 5 concludes the paper and refers to the future work.

2 The Instantaneous Signal Calculus

In this section, we introduce the instantaneous signal calculus which contains all conceptually instantaneous reactions of signal calculus for event-based synchronous languages. In the following part, all reactions are instantaneous, i.e., zero time reactions, if we do not mention it deliberately.

2.1 Signals, Events and Event Guards

We first give a brief introduction to signals, events and event guards for the definition of the instantaneous signal calculus. Technically, signals are means of communications and synchronisations between different parts of systems (agents) and between a agent and its environment. In our framework, we confine ourselves to pure signals which only carry the *present* or *absent* information of signals for the purposed of precise definition and mathematical treatment.

In general, a signal denoted by its name has three types of status, i.e., presence (+), absence (-) and unknown (0) [1]. The unknown status indicates that the present or absent information is still unclear at the time the observation is made, but it may be replaced by a presence or absence status when the reactions perform their behaviors. Given a signal s , we use tuples $(s, +)$, $(s, -)$, $(s, 0)$ to represent the presence, absence and unknown status respectively. Note that the status of signal should be consistent within its visible spatial range. Besides, the status of signal is transient from a time perspective, i.e., the status of signal is determined on a per-instant basis. Obviously, the status of signal at different instants are usually different. The status of signal in the current instant is not relevant to the one in the next instant.

Given a set S of signals, also called a *sort*, an (ordinary) event e specifies the status of signals in the sort. The notation $\text{sort}(e)$ denotes the sort of event e . It describes the state of the system and its environment. Formally, an event can be modeled as a function from its sort S to the consistent set $\mathcal{B} = \{+, -, 0\}$ which is a Scott flat Boolean domain [13], ordered by the relation $\{+ \geq 0, - \geq 0\}$. Alternatively, an event is also considered as a set of signals with status, i.e., for each signal, its status is unique in the set. In technical, all the signals having unknown status can be omitted in the set. For instance, given an event $e = \{(t, +)\}$ and its sort S , any signal s different from signal t has the unknown status in the event e , i.e., $\forall s \in S \bullet s \neq t \bullet (s, 0) \in e$. Thus $e(s) = +$ and $(s, +) \in e$ are both allowed to indicate the presence status of signal s in event e . The notations for the absence or unknown status are similar. The notation $\mathbb{E}(S)$ stands for the set of all events having sort S .

Definition 1 (Compatible). Let e_1 and e_2 be events over sort S , events e_1 and e_2 are compatible if it is of no conflict on the status in the sort, i.e., $\forall s \in S \bullet e_1(s) = e_2(s) \vee (s, 0) \in e_1 \vee (s, 0) \in e_2$. We denote it by $\text{compatible}(e_1, e_2)$.

For incompatible events e_1 and e_2 , there exists at least a signal s such that the status of s in the two events are conflict, i.e., $e_1(s) = + \wedge e_2(s) = - \vee e_1(s) = - \wedge e_2(s) = +$. In

¹ As a variant of signal model in [16], the three-value signal model preserves the algebraic semantics of instantaneous signal calculus.

general, the unknown status is not in conflict with the presence or absence status since it may be replaced by the other two status.

Here, we extend the set $\mathbb{E}(S)$ of events by a special element \perp to $\mathbb{E}^\perp(S)$. For any signal s , the status in the event \perp is meaningless. The event \perp indicates the system leads to a chaotic state. Two types of order relation over $\mathbb{E}^\perp(S)$ are introduced to semantic research later.

Definition 2 (Refinement). *Given events e_1 and e_2 in $\mathbb{E}^\perp(S)$, event e_1 is refined by event e_2 , denoted by $e_1 \leq e_2$, if either e_2 is event \perp or for each s , the status in the event e_2 is not less than the one in e_1 , i.e., $e_2 = \perp \vee \forall s \in S \bullet e_2(s) \geq e_1(s)$.*

The refinement relation indicates that for the observation of signal status, e_2 has more information than e_1 . Obviously, \perp is the greatest event under the refinement.

Definition 3 (Strong Refinement). *Given events e_1 and e_2 in $\mathbb{E}^\perp(S)$, event e_1 is strongly refined by event e_2 , denoted by $e_1 \leq_1 e_2$, if either e_2 is event \perp or events e_1 and e_2 have the same set of signals with absence status and the latter owns a greater set of signals with presence status, i.e., $e_2 = \perp \vee \forall s \in S \bullet e_2(s) = e_1(s) \vee e_1(s) = 0 \wedge e_1(s) = +$.*

As described in definition 3, the strong refinement reveals the relation of the set of present signals between events having the same set of signals with absence status. Naturally, \perp is also the greatest event under the strong refinement. Note that the strong refinement relation is the subset of the refinement relation, i.e., if event e_1 strongly refine e_2 , we also have $e_1 \geq e_2$.

Finally, we introduce several operators on events. The notation $e \oplus (s, b)$ stands for the update of e by the tuple (s, b) , where e is an ordinary event and $b \in \mathcal{B}$. For any signal t ($t \neq s$), the status in event $e \oplus (s, b)$ is the same as the one in e ; for signal s , $(s, b) \in e \oplus (s, b)$. Given an event e , we write $e \setminus s$ to denote the event of sort $\text{sort}(e) \setminus s$ which coincides with e on all signals but s . In particular, $\perp \setminus s =_{df} \perp$. Further, given events e_1 and e_2 over S , the merge $e_1 \cup e_2$ generates a new event. When e_1 and e_2 are compatible, for any signal $s \in S$, $(e_1 \cup e_2)(s) = \max(e_1(s), e_2(s))$; otherwise, $e_1 \cup e_2$ leads to the event \perp .

As the component of reactions, the event guards synchronise the behaviors of agents. When the event can trigger the event guard of a reaction, the reaction is enabled. The notations of event guards are given as follows:

$$g ::= \epsilon \mid \emptyset \mid s^+ \mid s^- \mid g \cdot g \mid g + g \mid \bar{g}$$

Now we give the meanings of event guards in Table 1. Let \mathbb{E} be the set of all ordinary events over the same sort. In actual, an event guard is identified as a set of ordinary events which can trigger the guard. Almost all event guards have the usual meanings and the definitions are straightforward. Guard ϵ can be triggered by every event in \mathbb{E} . By contrast, no any event can trigger the guard \emptyset . Guard s^+ defines the set of events in which the status is present while s^- represents the set of events in which the status is absent. Guard $g_1 \cdot g_2$ can be fired by the event which triggers both g_1 and g_2 . The event firing either g_1 or g_2 can trigger guard $g_1 + g_2$. Intuitively, \bar{g} defines all events which are incompatible of any event in g .

Table 1. The Meanings of Event Guards

| |
|--|
| $\begin{aligned} \llbracket \epsilon \rrbracket &=_{df} \mathbb{E} & \llbracket \emptyset \rrbracket &=_{df} \emptyset & \llbracket s^+ \rrbracket &=_{df} \{e \mid (s, +) \in e \wedge e \in \mathbb{E}\} \\ \llbracket s^- \rrbracket &=_{df} \{e \mid (s, -) \in e \wedge e \in \mathbb{E}\} & \llbracket g_1 + g_2 \rrbracket &=_{df} \llbracket g_1 \rrbracket \cup \llbracket g_2 \rrbracket \\ \llbracket g_1 \cdot g_2 \rrbracket &=_{df} \{e \mid e \in \llbracket g_1 \rrbracket \wedge e \in \llbracket g_2 \rrbracket\} \\ \llbracket \overline{g} \rrbracket &=_{df} \{e \mid \forall e' \in \llbracket g \rrbracket \bullet \neg \text{compatible}(e, e')\} \end{aligned}$ |
|--|

2.2 Instantaneous Reactions

Here, we give the syntax of the instantaneous reactions below.

$$I ::= !s \mid II \mid \perp \mid g\&I \mid I \setminus s \mid I; I \mid I \parallel I$$

Informally, each reaction may sense the status of signals in the input event and generate the output event. Due to the so-called synchronous hypothesis, outputs are generated at the same instant inputs are observed. The inputs as well as the generated outputs broadcast to all agents instantaneously. The outputs can be generated simultaneously by several agents.

The meanings of all reactions are accord with the common intuitions. The reaction $!s$ emits signal s and terminates immediately; II does nothing but terminates successfully. \perp represents the worst reaction which leads to a chaotic state. The reaction $g\&I$ behaves like I when the guard g is fired, otherwise it behaves like the reaction II . The reaction $I \setminus s$ declares signal s as a local signal and the emission of s becomes invisible to outside. $I_1; I_2$ indicates the sequential composition. $I_1 \parallel I_2$ immediately starts I_1 and I_2 in parallel. Note that I_1 and I_2 can interact with each other. Two examples are given to illustrate the execution of reactions.

Example 1. Let $I_1 = s_1^+ \&!s_2 \parallel (s_2^+ \cdot s_1^+) \&!s_3 \parallel s_2^- \&\perp$.

Given an input event $e_1 = \{(s_1, +)\}$, the guard s_1^+ can be triggered; thus signal s_2 is emitted immediately; at the time, the guard $s_2^+ \cdot s_1^+$ is also satisfied and then s_3 is generated. Hence, I_1 would react to the input event e_1 by emitting s_2 and s_3 . For input event $e_2 = \{(s_2, -)\}$, I_1 becomes chaotic since s_2 is absent in the input event and no reaction can emit it.

Example 2. Let $I_2 = s_1^+ \&!s_2 \parallel (s_2^+ \cdot s_1^+) \&!s_3 \parallel s_3^+ \&\perp$.

Given an input event $e_1 = \{(s_1, +)\}$, intuitively both signals s_2 and s_3 will be emitted according to the above computation. However the reaction actually enters into chaos state since s_3^+ activates the reaction \perp .

As can be seen from these examples, the computation of an reaction is proceeded step by step. When input event is given, we first inspect which guards are triggered. If the guard is fired, the involved reaction will generate the corresponding signals. Then with the generated signals, we repeat the computation until no new guard can be fired.

3 The Observation-Oriented Denotational Semantics

This section considers the observation-oriented semantics model for instantaneous reactions. We first investigate the *healthiness conditions*, especially dealing with the emission of signals. Healthiness conditions identify the valid semantic domains that characterize the instantaneous reactions. Then the operators and the corresponding closure are discussed to present the denotational semantic definitions. Finally, we define the semantics of primitives and combinators concerning instantaneous reactions.

3.1 Healthiness Conditions

As described above, a reaction reacts to an input event by computing an output event, i.e., by assigning a status to each signal. Intuitively, each reaction can be formalized as an function over events. In other words, for each reaction, for a given set of input signals with status, it generates a unique set of output signals with status. Note that the input event and output event should defined over the same sort. However, not all such functions can be denoted by significant reactions since all reactions actually tend to generate signals. Thus, inspired by Hoare & He's work in [5], healthiness conditions are employed to characterize a valid semantic domain including all healthiness functions over events each of which can be denoted by an instantaneous reaction. Thus we first focus on the definitions which are necessary for the valid domain.

Definition 4. *An (event) function over events $\mathbb{E}^\perp(A)$ maps each event in $\mathbb{E}^\perp(A)$ to the event in $\mathbb{E}^\perp(A)$, i.e., $f : \mathbb{E}^\perp(A) \rightarrow \mathbb{E}^\perp(A)$, where the sort A stands for the alphabet of function f , denoted by $\alpha f = A$.*

On the basis of the order relations over events, the corresponding order over functions can be easily derived, i.e., $f_1 \geq f_2 =_{df} \forall e \bullet f_1(e) \geq f_2(e)$ and $f_1 \geq_1 f_2 =_{df} \forall e \bullet f_1(e) \geq_1 f_2(e)$. Similarly, we have $f_1 \geq_1 f_2 \Rightarrow f_1 \geq f_2$.

Definition 5 (Strengthening). *Given a function f , we say that f is strengthening, if $\forall e \in \alpha f \bullet f(e) \geq e$.*

Obviously, strengthening functions improve the input event and those signals with unknown status may be updated into the presence/absence status. The status of signals which are not originally unknown can not be altered.

Definition 6 (Strongly Strengthening). *Given a function f , we say that f is strongly strengthening, if $\forall e \in \alpha f \bullet f(e) \geq_1 e$.*

Recall that all reactions tend to generate signals. The strongly strengthening functions indicate that the new signals are generated in the output event. Note that both the input event and the output event have the same set of signals with absence status. Actually, the strongly strengthening function is also strengthening, but not vice versa.

Definition 7 (Strictness). *Given a function f , we say that f is strict, if $f(\perp) = \perp$.*

We expect a strict function to behave chaotically if its initial input event is divergent, and the result of executing it could lead to the chaotic state.

Theorem 1. *A function f is strict if it is strengthening.*

The proof may be easily validated according to the definitions .

Definition 8 (Monotonic). *Given a function f , we say that f is monotonic, if $\forall e_1, e_2 \in \alpha f \bullet e_1 \geq e_2 \Rightarrow f(e_1) \geq f(e_2)$.*

The monotonic functions preserve the refinement relation of events, i.e., given a finer input event, it can always generate a finer output event.

Definition 9 (Strongly Monotonic). *Given a function f , we say that f is strongly monotonic, if $\forall e_1, e_2 \in \alpha f \bullet e_1 \geq_1 e_2 \Rightarrow f(e_1) \geq_1 f(e_2)$.*

Similarly, the strongly monotonic functions preserve the strong refinement relation over events. Given more present signals, it generates more output signals.

Theorem 2. *A function f is strongly monotonic if it is strongly strengthening and monotonic.*

Proof. Given events e_1, e_2 . Suppose $e_1 \geq_1 e_2$. Obviously, we have $e_1 \geq e_2$. Thus $f(e_1) \geq f(e_2)$ is validated since f is monotonic. Besides, due to the strongly strengthening property of function f , both $f(e_1) \geq_1 e_1$ and $f(e_2) \geq_1 e_2$ are obtained. If $f(e_1)$ is the chaotic event \perp , it is obvious that $f(e_1) \geq_1 f(e_2)$. Otherwise, suppose $f(e_1) \neq \perp$. Thus we get $\forall s \in \alpha f \bullet f(e_1)(s) \geq f(e_2)(s)$. For each signal s , if $(s, -) \in f(e_1)$ is valid, $(s, -) \in e_1$ is derived from the inequality $f(e_1) \geq_1 e_1$. Similarly, we also have $(s, -) \in e_2$ and $(s, -) \in f(e_2)$ since both $e_1 \geq_1 e_2$ and $f(e_2) \geq_1 e_2$ are validated; that is $\forall s \in \alpha f \bullet (s, -) \in f(e_1) \Rightarrow (s, -) \in f(e_2)$. Consequently, $f(e_1) \geq f(e_2)$ is obtained, i.e., function f is strongly monotonic.

Definition 10 (Idempotent). *Given a function f , we say f is idempotent, if $f \circ f = f$.*

From the definition, it is obvious that the idempotent function is stable under its output events, i.e., $f(e)$ is the fix point of function f . The generated signals can not give rise to the response. In other words, the response to the output event is trivially.

Now we give the definition for healthiness functions:

Definition 11 (Healthiness Functions). *Given a function f , we say that f is healthy, if it is strongly strengthening, monotonic, idempotent.*

Obviously, healthiness functions are also strict and strongly monotonic according to the theorem [11](#) and [2](#). Due to the synchronous hypothesis, we expect an instantaneous reaction to denote a healthiness function which is strengthening, monotonic, idempotent, strict and strongly monotonic.

For each instant, the status of signals should be unique and stable; that is, the denoted functions are expected to be idempotent. Further, the functions should be strengthening since all reactions tend to generate new signals. Consequently, the strictness is derived and it indicates that for each reactions, the response to the chaotic event leads to the chaotic result. Finally, it is obvious that finer input events can always give rise to the generation of finer output events, i.e., the functions should be monotonic and strongly monotonic.

3.2 The Semantic Domain

Now we turn to the definition of semantic domain, denoted by \mathcal{I} , which including all healthiness functions. Several primitive healthiness functions are also constructed. Besides, two operators over \mathcal{I} including *composition* (\circ), *addition* (\cup) are investigated for the semantic definition later. However, not all the operators preserve the closure with respect to the healthiness conditions. Thus the corresponding closure concerning the operators are investigated and the least fix-point is also introduced to regain the healthiness.

Firstly, we introduce primitive function χ_s which generates signal s as follows.

Definition 12 (Signal Emission)

$$\chi_s(e) =_{df} \begin{cases} e \oplus (s, +) & : e(s) \neq - \wedge e \neq \perp \\ \perp & : e(s) = - \vee e = \perp \end{cases}$$

Note that the event update function of first branch is strongly strengthening. As a constant function, the second branch is also a strongly strengthening function. Hence, the function χ_s is strongly strengthening and therefore it is also strict.

Given events e_1, e_2 and $e_1 \geq e_2$, if $(s, -) \in e_1$, $\chi_s(e_1) = \perp$, then $\chi_s(e_1) \geq \chi_s(e_1)$; if $(s, -) \notin e_1$, thus we have $(s, -) \notin e_2$, $\chi_s(e_1) = e_1 \oplus (s, +) \geq e_2 \oplus (s, +) = \chi_s(e_2)$. Hence, we have that χ_s is monotonic.

Further, the event update function of first branch is idempotent, and the output event is still within the domain of the first branch. It is the same as the constant function. Thus the function χ_s is idempotent.

As described above, χ_s should be a healthiness function. Furthermore, let S be a finite set of signals, we define $\chi_S =_{df} \chi_{S \setminus \{s\}} \circ \chi_s$. In particular, if $S = \emptyset$, define $\chi_\emptyset(e) =_{df} e$ for each event e . Obviously, function χ_S is strongly strengthening, strict, idempotent and monotonic. Due to the unordered of set, the order of emission of signals can be interchangeable which certifies the soundness of the definition. Thus function χ_S is called the signal emission function hereinafter.

Theorem 3. *The signal emission functions can be used to specify instantaneous reaction, i.e., $\chi_S \in \mathcal{I}$.*

We write the divergent function as χ_\perp ($\forall e \bullet \chi_\perp(e) = \perp$), which generates divergent event \perp on any input event.

Theorem 4. *The divergent function can be used to specify instantaneous reaction, i.e., $\chi_\perp \in \mathcal{I}$.*

Now, we introduce two operators over \mathcal{I} including \circ and \cup for deriving functions from primitives. For composite operator, $f_1 \circ f_2(e) =_{df} f_1(f_2(e))$. Easily, we claim that composite functions preserve the monotonicity, strong monotonicity and strong strengthening, taking advantage of the definition. Besides, composite healthiness functions are idempotent if they are commutative according to the lemma below.

Lemma 1. *Given functions $f_1, f_2 \in \mathcal{I}$, the composite functions $f_1 \circ f_2, f_2 \circ f_1$ are both healthy if f_1 and f_2 is commutative, i.e., $f_1 \circ f_2 = f_2 \circ f_1$.*

Next, we give the definition of addition over the semantic domain \mathcal{I} .

Definition 13. Given functions f and g , define $(f \cup g)(e) =_{df} f(e) \cup g(e)$.

From the definition, the addition is commutative and associative, taking advantage of the properties of events. Given a countable index set K , the notation $\bigcup_{k \in K} f_k$ represents the addition over a countable set of functions.

Lemma 2. Given strongly strengthening functions f_1 and f_2 , the formula $f_1 \cup f_2 \geq_1 f_1$ is validated.

The lemma [2] indicates that the addition preserves the strongly strengthening property, and thus it also maintains the strictness of functions. Furthermore, $f_1 \cup f_2$ is also monotonic if both f_1 and f_2 are monotonic. Unfortunately, the addition, in general, may not preserve the closure of the idempotence. Hence, we introduce the recursive computation to regain the expected idempotence. In actual, the recursive computation refers to the least fix-point theory explored below.

Definition 14 (Continuity). Given a set \mathcal{F} of event functions, A function $F : \mathcal{F} \rightarrow \mathcal{F}$ is said to be continuous if it distributes over the limits of all chains of strongly strengthening event functions, i.e., for each chain f_i ($i \geq 0$) such that $f_{k+1} \geq_1 f_k$ ($k \geq 0$) is validated, $F(\bigcup_i f_i) = \bigcup_i F(f_i)$.

In general, we say function $F : \mathcal{F} \rightarrow \mathcal{F}$ is monotonic if for each monotonic event function, it generates a monotonic function. For other properties of event functions, the definitions are similar. Obviously, a continuous function always yields to a strongly monotonic function.

Lemma 3. Given strongly strengthening and continuous function F , $\mu X \bullet F(X) = \bigcup_{n \geq 0} F^n(\chi_\emptyset)$.

where, $F^0(X) =_{df} \chi_\emptyset$, $F^{n+1}(X) =_{df} F(F^n(X))$.

This lemma is a variant of Kleene's fix point theory [15][5] and the proof is similar.

Corollary 1. Given strongly strengthening function f , $F(X) = f \circ X$ is continuous and $\mu X \bullet (f \circ X) = \bigcup_{n \geq 0} f^n$.

Theorem 5. Given monotonic and strongly strengthening function f and $F(X) = f \circ X$, $\mu X \bullet F(X) = \bigcup_{n \geq 0} f^n$ is idempotent, i.e., $\mu X \bullet F(X) \in \mathcal{I}$.

Proof. Easily, f^i ($i \geq 0$) is a chain of strongly strengthening functions. Thus we have $\bigcup_{n \geq 0} f^n = \lim_{i \rightarrow \infty} f_i$. For each event, it has a finite sort. Hence, The set of all events is also finite. Consequently, there exists $M \geq 0$ such that $\forall k \geq M \bullet \bigcup_{n \geq 0} f^n = f^k$. Thus $\bigcup_{n \geq 0} f^n$ is monotonic and strongly strengthening since composite functions preserve the monotonicity and strongly strengthening. For each event e , we obtain $(\bigcup_{n \geq 0} f^n \circ \bigcup_{n \geq 0} f^n)(e) = f^{2M}(e) = (\bigcup_{n \geq 0} f^n)(e)$, i.e., $\bigcup_{n \geq 0} f^n$ is idempotent. Thus $\bigcup_{n \geq 0} f^n \in \mathcal{I}$ is validated.

Theorem [5] indicates that the recursive computation can regain the idempotence according to the least fix-point theory. Recall that both the composition and the addition preserve the monotonicity and strongly strengthening. Thus the least fix-point theory provides an approach to generating healthiness functions by the recursive computation.

3.3 The Observation-Oriented Semantics

Here, we propose the observation-oriented semantics for each instantaneous reactions, i.e., each reaction is formally identified as a healthiness function. We write $\llbracket I \rrbracket$ to represent the denotational semantic definition for reaction I .

For each instantaneous reaction I and event e , we firstly define (semantic) *output* set regarding e and (semantic) *divergent* condition in the framework of the denotational semantic model.

Definition 15. *Given reaction I and event e ($e \neq \perp$), the output set, denoted by $\text{output}(I, e) =_{df} \{s \mid (s, 0) \in e \wedge (s, +) \in \llbracket I \rrbracket(e)\}$, embodies all signals generated by reaction I regarding event e . The divergent condition, denoted by $\text{div}(I) =_{df} \{e \mid \llbracket I \rrbracket(e) = \perp\}$, consists of all the events which leads to the chaotic state.*

Now, the semantics of all primitives are defined as follows.

Definition 16 (Primitives). *The reactions $!s$, $!!$ and \perp are defined as $\llbracket !s \rrbracket =_{df} \chi_s$, $\llbracket !! \rrbracket =_{df} \chi_\emptyset$, and $\llbracket \perp \rrbracket =_{df} \chi_\perp$ respectively.*

Next, we give the semantic definitions for all combinators.

Definition 17 (Guarded Reactions)

$$\llbracket g \& P \rrbracket(e) =_{df} \begin{cases} \llbracket P \rrbracket(e) & : e \in \llbracket g \rrbracket \wedge e \neq \perp \\ e & : e \notin \llbracket g \rrbracket \wedge e \neq \perp \\ \perp & : e = \perp \end{cases}$$

Clearly, all guarded reactions should be idempotent, monotonic and strongly strengthening. For each branch, the corresponding function is idempotent and strongly strengthening and the output event is still within the domain of function. Thus the piecewise function is also idempotent and strongly strengthening. For the monotonicity, given events e_1, e_2 and $e_1 \geq e_2$. If $e_1 = \perp \vee e_2 = \perp$, we have $\llbracket g \& P \rrbracket(e_1) = \perp$. Otherwise, consider $e_2 \neq \perp$, if e_2 can trigger guard g , e_1 is also enabled to trigger g . Thus $\llbracket g \& P \rrbracket(e_1) \geq \llbracket g \& P \rrbracket(e_2)$ is obtained following the monotonicity of $\llbracket P \rrbracket$; if e_2 can not trigger g , we get that $\llbracket g \& P \rrbracket(e_2) = e_2$ from the definition. According to the strengthening of $\llbracket g \& P \rrbracket$, $\llbracket g \& P \rrbracket(e_1) \geq e_1 \geq e_2 = \llbracket g \& P \rrbracket(e_2)$ is derived. Hence we claim that $\llbracket g \& P \rrbracket$ is monotonic, i.e., $\llbracket g \& P \rrbracket \in \mathcal{I}$.

Definition 18 (Concealment)

$$\llbracket P \setminus s \rrbracket(e) =_{df} \begin{cases} \llbracket P \rrbracket(e \oplus (s, +)) \setminus s & : \llbracket P \rrbracket(e)(s) = + \\ \llbracket P \rrbracket(e \oplus (s, -)) \setminus s & : \llbracket P \rrbracket(e)(s) \neq + \wedge \forall e' \geq e \bullet \llbracket P \rrbracket(e')(s) \neq + \\ \llbracket P \rrbracket(e) & : \llbracket P \rrbracket(e)(s) \neq + \wedge \exists e' \geq e \bullet \llbracket P \rrbracket(e')(s) = + \\ \perp & : \llbracket P \rrbracket(e) = \perp \end{cases}$$

Similarly, the concealment reactions are also idempotent, monotonic and strongly strengthening. In general, if reaction P inside can generate signal local signal s , we obtain the result by calculating the reaction to the new event $e \oplus (s, +)$. Naturally, the local signal s is always invisible to the outside. In contrast, if P can not emit s , two

possible cases are discussed. On one hand, if s cannot be emitted indeed even if the input event is enhanced, then the newly input event is replaced by $e \oplus (s, -)$; on the other hand, if s cannot be emitted due to the insufficiency of signal status, then the result of concealment should be identical with the result of P .

Definition 19 (Sequential)

$$\llbracket I_1; I_2 \rrbracket(e) =_{df} \begin{cases} \llbracket I_2 \rrbracket \circ \llbracket I_1 \rrbracket(e) & : e \in C_1(I_1) \\ \llbracket I_1 \rrbracket(e) & : e \in C_2(I_1) \end{cases}$$

where, $C_1(I) =_{df} \{e \mid \forall e' \geq e \bullet e' \neq \perp \Rightarrow \text{output}(I, e') = \text{output}(I, e)\}$, $C_2(I) =_{df} \{e \mid \exists e' \geq e \bullet \text{output}(I, e') \supseteq \text{output}(I, e)\}$.

Due to the unknown status, some events can neither trigger the guard nor reject the guard, e.g., event $e = \{(s, 0)\}$ and guard $g = s^+$. In this circumstance, due to the insufficiency of signal status, it may lead to the inadequate execution of reactions, which indicates that more signals can be generated if the input event is enhanced regarding the strengthening. Thus we should take the adequate execution of reaction into consideration when establishing the denotational semantics of the sequential. The status of output event and control are passed on to reaction I_2 when no more output signal can be generated even if the input event is improved. Otherwise the execution of reaction I_1 is inadequate, and the sequential $I_1; I_2$ behaves as I_1 exactly.

Definition 20 (Parallel)

$$\llbracket I_1 \parallel I_2 \rrbracket =_{df} \mu X \bullet (\llbracket I_1 \rrbracket \cup \llbracket I_2 \rrbracket) \circ X$$

Intuitively, $I_1 \parallel I_2$ immediately starts I_1 and I_2 in parallel. The function $\llbracket I_1 \rrbracket \cup \llbracket I_2 \rrbracket$ seems as an appropriate candidate. Unfortunately, it can not preserve the idempotence according to the discussion above. Thus the recursion is employed to regain the healthiness based on the least fix-point theory.

Corollary 2. $\llbracket \bigcup_{m \in M} P_m \rrbracket = \mu X \bullet (\bigcup_{m \in M} \llbracket P_m \rrbracket) \circ X$.

Definition 21 (Normal Form). *The reaction $\llbracket \bigcup_{m \in M} g_m \&!s_m \rrbracket \parallel h \&\perp$ is a normal form for instantaneous reactions if it satisfies the two conditions below, where all g_i and h are guards, the index set M is finite and all signals s_i ($i \in M$) are different.*

- (1). $\forall m, n \in M, g \bullet (g \cdot s_n^+ \subseteq g_m \Rightarrow g \cdot g_n \subseteq g_m) \wedge (g \cdot s_n^+ \subseteq h \Rightarrow g \cdot g_n \subseteq h)$.
- (2). $\forall m \in M, g_m \cdot s_m^- \subseteq h \subseteq g_m \wedge s_m^+ \subseteq g_m$.

Theorem 6. *All instantaneous reactions can be reduced into normal forms.*

Theorem 6 indicates that every instantaneous reaction, however deeply structured, can be reduced into the normal form by a series of algebraic manipulation. The proof is given in [16].

Theorem 7. *Given reaction $I = \llbracket \bigcup_{m \in M} g_m \&!s_m \rrbracket \parallel h \&\perp$ in normal form, we have*

$$\llbracket I \rrbracket = \bigcup_{i \in M} \llbracket g_m \&!s_m \rrbracket \cup \llbracket h \&\perp \rrbracket.$$

As shown in theorem 7 the normal form, surprisingly, has the comparatively elegant and straightforward denotational semantic definition since the recursion is avoided. Considering theorem 6 and theorem 7 we actually obtain an effective approach to the semantic definitions for instantaneous reactions, by transforming algebraically all reactions into normal form and calculating the semantics for normal form reactions. From the practical point of view, semantic definitions achieved by term rewriting rather than direct semantic computing (including recursion) is amenable to efficient implementation by computer.

Finally, we define the equivalence between instantaneous reactions based on the denotational semantic model.

Definition 22. We say two instantaneous reactions I_1 and I_2 are (semantically) equivalent regarding the denotational semantic model if $\llbracket I_1 \rrbracket = \llbracket I_2 \rrbracket$, denoted by $I_1 =_D I_2$.

4 Algebraic Properties

Algebra is well-suited for direct use by engineers in symbolic calculation of parameters and the structure of an optimal design. Algebraic proof by term rewriting is the most promising way in which computers can assist in the process of reliable design. In this section, we propose a set of algebraic laws concerning the distinct features for instantaneous reactions. All algebraic laws can be established in the framework of our denotational model, i.e., if the equality of two differently written instantaneous reactions is algebraically provable, the two reactions are also equivalent with respect to the denotational semantics. Due to the space limit, proofs that the algebraic laws are sound with respect to the denotational semantics are straightforward and have been omitted.

4.1 Parallel

The parallel is commutative and associative. Consequently, the order of parallel composition is irrelevant.

$$\text{par - 1} \quad I_1 \parallel I_2 =_D I_2 \parallel I_1 \quad (\parallel \text{ comm})$$

$$\text{par - 2} \quad (I_1 \parallel I_2) \parallel I_3 =_D I_1 \parallel (I_2 \parallel I_3) \quad (\parallel \text{ assoc})$$

The parallel is idempotent, due to deterministic behavior of reactions.

$$\text{par - 3} \quad I \parallel I =_D I \quad (\parallel \text{ idemp})$$

Reactions \perp and II are the zero and the unit of parallel composition respectively.

$$\text{par - 4} \quad \perp \parallel I =_D \perp \quad (\parallel - \perp \text{ zero})$$

$$\text{par - 5} \quad II \parallel I =_D I \quad (\parallel - II \text{ unit})$$

A guard g also triggers the reaction $s^+ \& I$ if it can generate signal s .

$$\text{par - 6} \quad g \& !s \parallel s^+ \& I =_D g \& !s \parallel (s^+ + g) \& I$$

4.2 Guard

The following law enables us to eliminate nested guards.

$$\mathbf{guard - 1} \quad g_1 \& (g_2 \& I) =_D (g_1 \cdot g_2) \& I \quad (\& \text{ multi})$$

Event guards with same reaction can be combined.

$$\mathbf{guard - 2} \quad g_1 \& I \parallel g_2 \& I =_D (g_1 + g_2) \& I \quad (\& \text{ add})$$

The event guard distributes through the parallel.

$$\mathbf{guard - 3} \quad g \& (I_1 \parallel I_2) =_D g \& I_1 \parallel g \& I_2 \quad (\& - \parallel \text{ distrib})$$

Reaction $\emptyset \& I$ behaves like I because its guard can never be fired.

$$\mathbf{guard - 4} \quad \emptyset \& I =_D I \quad (\& - \emptyset \text{ top})$$

Reaction $\epsilon \& I$ always activates the reaction I .

$$\mathbf{guard - 5} \quad \epsilon \& I =_D I \quad (\& - \epsilon \text{ bottom})$$

Reaction $g \& I$ never emits signals.

$$\mathbf{guard - 6} \quad g \& I =_D I \quad (\& - I \text{ void})$$

4.3 Sequential

Reaction I is the unit of sequential composition.

$$\mathbf{seq - 1} \quad I; I =_D I =_D I; I \quad (; - \text{unit zero})$$

The sequential distributes forwards through the parallel.

$$\mathbf{seq - 2} \quad I_1; I_2 \parallel I_3 =_D (I_1; I_2) \parallel (I_1; I_3) \quad (\parallel - I \text{ unit})$$

The reaction I can be executed only if the guard \bar{g} is triggered.

$$\mathbf{seq - 3} \quad g \& \perp; I =_D g \& \perp \parallel \bar{g} \& I$$

The following law enables us to convert the sequential into the parallel.

$$\mathbf{seq - 4} \quad g \& !s; h \& p =_D g \& !s \parallel (g \cdot (h[\epsilon/s^+, \emptyset/s^-]) + \bar{g} \cdot h) \& p$$

provided that $p \in \{\perp, !t, I\}$

4.4 Concealment

The concealment is commutative and the order is not critical.

$$\mathbf{conc - 1} \quad (I \setminus s) \setminus t =_D (I \setminus t) \setminus s \quad (\setminus \text{ comm})$$

$\setminus s$ distributes backward over \parallel when one component does not mention signal s .

conc - 2 $(I_1 \parallel I_2) \setminus s =_D (I_1 \setminus s) \parallel I_2$ provided that $s \notin I_2$ ($\setminus - \parallel$ *quasi-distrib*)

$\setminus s$ distributes backward over guarded reaction if s does not appear in the guard g .

conc - 3 $(g \& I) \setminus s =_D g \& (I \setminus s)$ provided that $s \notin g$ ($\setminus - \&$ *quasi-distrib*)

The following law illustrates how to eliminate the concealment.

conc - 4 $(g \&!s \parallel I) \setminus s =_D I[g/s^+, \bar{g}/s^-]$ provided that $s \notin g$ and $s \notin \text{ems}(I)$

where, $\text{ems}(I)$ defines the set of the possible signals generated by reaction I .

4.5 Primitives

When reaction $s^- \&!s$ is triggered it behaves like \perp since it violates the logical coherence between the environment assumptions (i.e., absence of signal s) and the effect of emission of signal s

prim - 1 $s^- \&!s =_D s^- \& \perp$ (*logical coherence*)

Reaction $s^+ \&!s$ behaves like II because emission of s does not change the statuses of s .

prim - 2 $s^+ \&!s =_D s^+ \& II$ (*axiom unit*)

5 Conclusions and Future Work

In this paper, we explore an observation-oriented denotational semantics for instantaneous signal calculus which contains all conceptually instantaneous reactions of signal calculus for event-based synchronous languages. Every instantaneous reaction is identified as denoting a healthiness function which is monotonic, strict, idempotent and strongly strengthening. Besides, a set of algebraic laws concerning the distinct features for instantaneous reactions are provided. All algebraic laws can be established in the framework of our denotational model.

In the future, we will integrate the top-down and the bottom-up methods on linking theories of programming by demonstrating the equivalence over denotational, algebraic and operational semantics. Further, we will complete the signal calculus by introducing delay-time reactions. We believe that the zero-time reactions and delay-time reactions are orthogonal. Thus the extension is straightforward.

Acknowledgement. This work was supported by National High Technology Research and Development Program of China (No. 2011AA010101 and and No. 2012AA011205), National Basic Research Program of China (No. 2011CB302904), National Natural Science Foundation of China (No. 61021004 and No. 90818024), and Shanghai Leading

Academic Discipline Project (No. B412). The authors also gratefully acknowledge support from the Danish National Research Foundation and the National Natural Science Foundation of China (Grant No. 61061130541) for the Danish-Chinese Center for Cyber Physical Systems.

References

1. Berry, G.: The foundations of estereel. In: Proof, Language, and Interaction, Essays in Honour of Robin Milner, pp. 425–454. The MIT Press (2000)
2. Berry, G., Gonthier, G.: The estereel synchronous programming language: Design, semantics, implementation. *Science Computer Program* 19(2), 87–152 (1992)
3. Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright, J.B.: Initial algebra semantics and continuous algebras. *Journal of the ACM* 24(1), 68–95 (1977)
4. Hoare, C.A.R., Hayes, I.J., He, J., Morgan, C., Roscoe, A.W., Sanders, J.W., Sørensen, I.H., Spivey, J.M., Suftrin, B.: Laws of programming. *Commun. ACM* 30(8), 672–686 (1987)
5. Hoare, C.A.R., He, J.: *Unifying Theories of Programming*. Prentice Hall International Series in Computer Science (1998)
6. Hoare, C.A.R., He, J., Sampaio, A.: Normal form approach to compiler design. *Acta Inf.* 30(8), 701–739 (1993)
7. Laplante, P.A.: *Real-Time Systems Design and Analysis: An Engineer's Handbook*, 2nd edn. IEEE Press (1997)
8. Lee, E.A.: *Cyber-Physical Systems - are computing foundations adequate?* Technical report, Department of EECS, UC Berkeley (2006)
9. Maddux, R.D.: Relation-algebraic semantics. *Theoretical Computer Science* 160(1&2), 1–85 (1996)
10. Mousavi, M.R.: Causality in the semantics of estereel: Revisited. In: *SOS: Structural Operational Semantics*. EPTCS, vol. 18, pp. 32–45 (2009)
11. Nissanke, N.: *Real time systems*. Prentice Hall series in computer science. Prentice Hall (1997)
12. Roscoe, A.W., Hoare, C.A.R.: The laws of occam programming. *Theoretical Computer Science* 60, 177–229 (1988)
13. Scott, D., Strachey, C.: Towards a mathematical semantics for computer languages. Technical Report PRG-6, Oxford University Computer Laboratory (1971)
14. Tardieu, O., de Simone, R.: Loops in estereel. *ACM Transaction in Embedded Computer Systems* 4(4), 708–750 (2005)
15. Tarski, A.: A lattice-theoretical fixpoint theorem and its application. *Pacific Journal of Mathematics* 5(2), 285–309 (1955)
16. Zhao, Y., Jifeng, H.: Towards a Signal Calculus for Event-Based Synchronous Languages. In: Qin, S., Qiu, Z. (eds.) *ICFEM 2011*. LNCS, vol. 6991, pp. 1–13. Springer, Heidelberg (2011)

A Timed Mobility Semantics Based on Rewriting Strategies

Gabriel Ciobanu¹, Maciej Koutny², and Jason Steggles²

¹ Faculty of Computer Science, A.I. Cuza University of Iasi, 700483 Iasi, Romania
gabriel@info.uaic.ro

² School of Computing Science, University of Newcastle, U.K.
{maciej.koutny, jason.steggles}@ncl.ac.uk

Abstract. We consider TiMo (Timed Mobility) which is a process algebra for prototyping software engineering applications supporting mobility and timing constraints. We provide an alternative semantics of TiMo using rewriting logic; in particular, we develop a rewriting logic model based on strategies to describe a maximal parallel computational step of a TiMo specification. This new semantical model is proved to be sound and complete w.r.t. to the original operational semantics which was based on negative premises. We implement the rewriting model within the strategy-based rewriting system ELAN, and provide an example illustrating how a TiMo specification is executed and how a range of (behavioural) properties are analysed.

Keywords: process algebra, mobility, time, rewriting logic, strategies.

1 Introduction

TiMo (Timed Mobility) is a process algebra proposed in [8] for prototyping software engineering applications in distributed system design. TiMo supports process mobility and interaction, and allows one to add timers to the basic migration and communication actions. Recently, the model has been extended to model security aspects such as access permissions [10]. The behaviour of TiMo specifications can be captured using a set of SOS rules or suitable Petri nets [9], both based on executing time actions with negative premises. In this paper, we provide an alternative semantics of TiMo using rewriting logic and strategies. Our aim is to obtain a semantical model of TiMo which can be used as the basis for developing efficient tool support and investigating different semantic choices.

Rewriting Logic (RL) [18] is an algebraic formalism for dynamic systems which uses equational specifications to define the states of a system, and rewrite rules to capture the dynamic state transitions. Strategies [5,6] are an integral part of RL which provide control over the rewriting process, allowing important dynamic properties to be modelled. In our work, we develop a RL model for TiMo specifications. In particular, we formulate a strategy which captures the maximal parallel computational step of a TiMo specification, including its time rule based on negative premises. The resulting RL model is then formally validated,

by showing that it is both *sound* and *complete* w.r.t. the original operational semantics of TiMO.

As a first attempt at developing tool support for TiMO based on the new semantics, we use the strategy-based rewrite system ELAN [4,6] to implement TiMO specifications. The simple example we discuss provides a useful insight into the proposed RL modelling approach, and illustrates the type of (behavioural) properties that can be analysed.

The paper is structured as follows. Section 2 describes the syntax and semantics of TiMO, and Section 3 briefly introduces RL and strategies. In Section 4, we develop an RL model of TiMO, and prove its correctness. In Section 5, we show how to implement our RL model within the ELAN, and discuss what properties can then be verified. Section 6 discusses related and future work.

2 TiMO (Timed Mobility Language)

TiMO (Timed Mobility) [8,9,10] is a process algebra for mobile systems where it is possible to add timers to the basic actions, and each location runs according to its own local clock which is invisible to processes. Processes have communication capabilities which are active up to a predefined time deadline. Other timing constraints specify the latest time for moving between locations.

We assume suitable data types together with associated operations, including a set *Loc* of *locations*, a set *Chan* of *communication channels*, and a set *Id* of process identifiers, where each $id \in Id$ has arity m_{id} . We use \mathbf{x} to denote a finite tuple of elements (x_1, \dots, x_k) whenever it does not lead to a confusion.

The syntax of TiMO is given in Table 1, where P represents *processes* and N represents *networks*. Moreover, for each $id \in Id$, there is a unique process definition (DEF), where P_{id} is a process expression, the u_i 's are distinct variables playing the role of parameters, and the X_i^{id} 's are data types. In Table 1, it is assumed that: (i) $a \in Chan$ is a channel, and $t \in \mathbb{N} \cup \{\infty\}$ represents a timeout; (ii) each v_i is an expression built from data values and variables; (iii) each u_i is a variable, and each X_i is a data type; (iv) l is a location or a location variable; and (v) \textcircled{S} is a special symbol used to state that a process is temporarily 'stalled'.

The only variable binding construct is $a^{\Delta t} ? (\mathbf{u}; \mathbf{X}) \text{ then } P \text{ else } P'$ which binds the variables \mathbf{u} within P (but *not* within P'). We use $fv(P)$ to denote the free variables of a process P (and similarly for networks). For a process definition as in (DEF), we assume that $fv(P_{id}) \subseteq \{u_1, \dots, u_{m_{id}}\}$, and so the free variables of P_{id} are parameter bound. Processes are defined up to the alpha-conversion, and $\{v/u, \dots\}P$ is obtained from P by replacing all free occurrences of a variable u by v , etc, possibly after alpha-converting P in order to avoid clashes. Moreover, if \mathbf{v} and \mathbf{u} are tuples of the same length then $\{\mathbf{v}/\mathbf{u}\}P$ denotes $\{v_1/u_1, v_2/u_2, \dots, v_k/u_k\}P$.

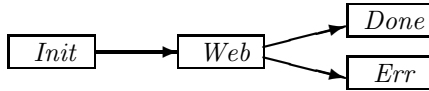
A process $a^{\Delta t} ! \langle \mathbf{v} \rangle \text{ then } P \text{ else } P'$ attempts to send a tuple of values \mathbf{v} over the channel a for t time units. If successful, it continues as process P ; otherwise it continues as the alternative process P' . A process $a^{\Delta t} ? (\mathbf{u}; \mathbf{X}) \text{ then } P \text{ else } P'$ attempts for t time units to input a tuple of values of type \mathbf{X} and substitute

Table 1. TiMo Syntax. Length of \mathbf{u} is the same as \mathbf{X} , and length of \mathbf{v} in $id(\mathbf{v})$ is m_{id} .

| | | |
|-------------------|--|----------------------|
| <i>Processes</i> | $P ::= a^{\Delta t} ! \langle \mathbf{v} \rangle \text{ then } P \text{ else } P' \mid$ | <i>(output)</i> |
| | $a^{\Delta t} ? (\mathbf{u} : \mathbf{X}) \text{ then } P \text{ else } P' \mid$ | <i>(input)</i> |
| | $\text{go}^{\Delta t} l \text{ then } P \mid$ | <i>(move)</i> |
| | $P \mid P' \mid$ | <i>(parallel)</i> |
| | $id(\mathbf{v}) \mid$ | <i>(recursion)</i> |
| | $\text{stop} \mid$ | <i>(termination)</i> |
| | $\textcircled{S} P$ | <i>(stalling)</i> |
| <i>Networks</i> | $N ::= l \llbracket P \rrbracket \mid N \mid N'$ | |
| <i>Definition</i> | $id(u_1, \dots, u_{m_{id}} : X_1^{id}, \dots, X_{m_{id}}^{id}) \stackrel{\text{df}}{=} P_{id}$ | (DEF) |

them for the variables \mathbf{u} . Mobility is implemented by a process $\text{go}^{\Delta t} l \text{ then } P$ which moves from the current location to the location l within t time units. Note that since l can be a variable, and so its value is assigned dynamically through communication with other processes, migration actions support a flexible scheme for moving processes around a network. Processes are further constructed from the (terminated) process stop and parallel composition $P \mid P'$. Finally, process expressions of the form $\textcircled{S} P$ are a purely technical device which is used in the subsequent formalisation of structural operational semantics of TiMo; intuitively, \textcircled{S} specifies that a process P is temporarily (i.e., until a clock tick) *stalled* and so cannot execute any action. A located process $l \llbracket P \rrbracket$ is a process running at location l , and a network is composed out of its components $N \mid N'$.

As an illustrative example, consider a simple workflow example in which a processing job moves from an initial location to a web service location and finally to a done location. If an error occurs with the web service then the job enters an error location. A pictorial representation of this example is given below.



The TiMo specification WF consists of four locations: *Init*; *Web*; *Done*; and *Err*. The following process identifier definitions are used:

$$\begin{aligned}
 job &\stackrel{\text{df}}{=} a^{\Delta 1} ? (l : Loc) \text{ then } \text{go}^{\Delta 1} l \text{ then } job \text{ else } job \\
 serv(l : Loc) &\stackrel{\text{df}}{=} a^{\Delta 2} ! \langle l \rangle \text{ then } serv(l) \text{ else } serv(l) \\
 servErr(l : Loc) &\stackrel{\text{df}}{=} a^{\Delta 2} ! \langle l \rangle \text{ then } servErr(l) \text{ else } servErr(Err)
 \end{aligned}$$

For instance, $Init \llbracket job \mid serv(Web) \rrbracket \mid Web \llbracket serv(Done) \rrbracket$ could be an initial TiMo network for this example.

A network N is *well-formed* if: (i) there are no free variables in N ; (ii) there are no occurrences of the special symbol \textcircled{S} in N ; (iii) assuming that id is as in the recursive equation (DEF), for every $id(\mathbf{v})$ occurring in N or on the right hand side

Table 2. Three rules of the structural equivalence (EQ1-EQ3), and six action rules (CALL), (MOVE), (COM), (PAR), (EQUIV), (TIME) of the operational semantics. In (PAR) and (EQUIV) ψ is an action, and in (TIME) l is a location.

$$\begin{array}{l}
\text{(EQ1-3)} \quad N | N' \equiv N' | N \quad (N | N') | N'' \equiv N | (N' | N'') \quad l \llbracket P | P' \rrbracket \equiv l \llbracket P \rrbracket | l \llbracket P' \rrbracket \\
\text{(CALL)} \quad l \llbracket id(\mathbf{v}) \rrbracket \xrightarrow{id @ l} l \llbracket \textcircled{\$} \{ \mathbf{v} / \mathbf{u} \} P_{id} \rrbracket \quad \text{(MOVE)} \quad l \llbracket go^{\Delta t} l' \text{ then } P \rrbracket \xrightarrow{l' @ l} l' \llbracket \textcircled{\$} P \rrbracket \\
\text{(COM)} \quad \frac{v_1 \in X_1 \dots v_k \in X_k}{l \llbracket a^{\Delta t} ! \langle \mathbf{v} \rangle \text{ then } P \text{ else } Q \mid a^{\Delta t'} ? (\mathbf{u} : \mathbf{X}) \text{ then } P' \text{ else } Q' \rrbracket} \\
\qquad \xrightarrow{a(\mathbf{v}) @ l} l \llbracket \textcircled{\$} P \mid \textcircled{\$} \{ \mathbf{v} / \mathbf{u} \} P' \rrbracket \\
\text{(PAR)} \quad \frac{N \xrightarrow{\psi} N'}{N | N'' \xrightarrow{\psi} N' | N''} \qquad \text{(TIME)} \quad \frac{N \not\xrightarrow{l}}{N \xrightarrow{\forall l} \phi_l(N)} \\
\text{(EQUIV)} \quad \frac{N \equiv N' \quad N' \xrightarrow{\psi} N'' \quad N'' \equiv N'''}{N \xrightarrow{\psi} N'''}
\end{array}$$

of any recursive equation, the expression v_i is of type corresponding to X_i^{id} . We let $Prs(TM)$ and $Net(TM)$ represent the set of well-formed TiMO process and network terms respectively. The first component of the operational semantics of TiMO is the structural equivalence \equiv on networks. It is the smallest congruence such that the equalities (EQ1–EQ3) in Table 2 hold. Using (EQ1–EQ3) one can always transform a given network N into a finite parallel composition of networks of the form $l_1 \llbracket P_1 \rrbracket \mid \dots \mid l_n \llbracket P_n \rrbracket$ such that no process P_i has the parallel composition operator at its topmost level. Each subnetwork $l_i \llbracket P_i \rrbracket$ is called a *component* of N , the set of all components is denoted by $comp(N)$, and the parallel composition is called a *component decomposition* of the network N . Note that these notions are well defined since component decomposition is unique up to the permutation of the components. This follows from the rule (CALL) which treats recursive definitions as function calls which take a unit of time. Another consequence of such a treatment is that it is impossible to execute an infinite sequence of action steps without executing any local clock ticks.

Table 2 introduces two kinds of operational semantics rules: $N \xrightarrow{\psi} N'$ and $N \xrightarrow{\forall l} N'$. The former is an execution of an action ψ by some process, and the latter a unit time progression at location l . In the rule (TIME), $N \not\xrightarrow{l}$ means that the rules (CALL) and (COM) as well as (MOVE) with $\Delta t = \Delta \theta$ cannot be applied to N for this particular location l . Moreover, $\phi_l(N)$ is obtained by taking the component decomposition of N and simultaneously replacing all the components of the form $l \llbracket go^{\Delta t} l' \text{ then } P \rrbracket$ by $l \llbracket go^{\Delta t-1} l' \text{ then } P \rrbracket$, and all components of the form $l \llbracket a^{\Delta t} \omega \text{ then } P \text{ else } Q \rrbracket$ (where ω stands for $! \langle \mathbf{v} \rangle$ or $? (\mathbf{u} : \mathbf{X})$) by $l \llbracket Q \rrbracket$ if $t = 0$, and $l \llbracket a^{\Delta t-1} \omega \text{ then } P \text{ else } Q \rrbracket$ otherwise. After that, all the occurrences of the symbol $\textcircled{\$}$ in N are erased.

The above defines executions of individual actions. A complete computational step is captured by a *derivation* of the form $N \xrightarrow{\Psi} N'$, where $\Psi = \{\psi_1, \dots, \psi_m\}$ ($m \geq 0$) is a finite multiset of l -actions for some location l (i.e., actions of the

form $id@l$ or $l'@l$ or $a\langle v \rangle@l$) such that $N \xrightarrow{\psi_1} N_1 \cdots N_{m-1} \xrightarrow{\psi_m} N_m \xrightarrow{\psi_l} N'$. That is, a derivation is a condensed representation of a sequence of individual actions followed by a clock tick, all happening at the same location. Intuitively, we capture the cumulative effect of the concurrent execution of the multiset of actions Ψ at location l . We say that N' is *directly reachable* from N . Note that whenever there is only a time progression at a location, we have $N \xrightarrow{\emptyset} N'$.

As an example, consider two derivation steps in the workflow network:

$$\begin{aligned} & \text{Init} \llbracket \text{job} \mid \text{serv}(\text{Web}) \rrbracket \mid \text{Web} \llbracket \text{serv}(\text{Done}) \rrbracket \xrightarrow{\{\text{job}@Init, \text{serv}@Init\}} \\ & \text{Init} \llbracket a^{\Delta 1} ?(l:\text{Loc}) \text{ then } \text{go}^{\Delta 1} l \text{ then } \text{job} \text{ else } \text{job} \mid a^{\Delta 2} !\langle \text{Web} \rangle \text{ then} \\ & \text{serv}(\text{Web}) \text{ else } \text{serv}(\text{Web}) \rrbracket \mid \text{Web} \llbracket \text{serv}(\text{Done}) \rrbracket \xrightarrow{\{a\langle \text{Web} \rangle@Init\}} \\ & \text{Init} \llbracket \text{go}^{\Delta 1} \text{Web} \text{ then } \text{job} \text{ else } \text{job} \mid \text{serv}(\text{Web}) \rrbracket \mid \text{Web} \llbracket \text{serv}(\text{Done}) \rrbracket \end{aligned}$$

One can show that derivations are well defined as one cannot execute an unbounded sequence of action moves without time progress, and the execution Ψ is made up of independent (or concurrent) individual executions. Moreover, derivations preserve well-formedness of networks (see [8]).

3 Rewriting Logic and Strategies

Rewriting logic (RL) [18] is an algebraic specification approach which is able to model dynamic system behaviour. In RL the static properties of a system are described by a standard algebraic specification, whereas the dynamic behaviour of the system is modelled using rewrite rules. Rewrite strategies are then used to control the application of rewrite rules and allow a RL specification to capture subtle aspects of a dynamic system. A brief introduction to RL and rewriting strategies is presented below (for a more detailed introduction see [18,6]).

An S -sorted signature Σ defines a collection of function symbols, where: $c : s \in \Sigma$ means c is a constant symbol of sort $s \in S$; and $f : s(1) \dots s(n) \rightarrow s \in \Sigma$ means f is a function symbol in Σ of *domain type* $s(1) \dots s(n)$, *arity* n , and *codomain type* s . Let $X = \langle X_s \mid s \in S \rangle$ be a family of sets of variables. We let $T(\Sigma, X) = \langle T(\Sigma, X)_s \mid s \in S \rangle$ be the family of sets of all terms over Σ and X . For any term $t \in T(\Sigma, X)_s$, we let $\text{Var}(t) \subseteq \cup_{s \in S} X_s$ represent the set of variables used in t . We let $T(\Sigma, X)/E$ represent the free quotient algebra of terms with respect to a set of equations E over Σ and X . For for any term $t \in T(\Sigma, X)_s$, we let $\langle t \rangle_E$ represent the equivalence class of term t with respect to the equations E (see [17]).

In RL a specification (Σ, E) defines the states $\langle t \rangle_E$ of a system. The dynamic behaviour of the system is then specified by *rewrite rules* [18,6]: $l \Longrightarrow r$, for terms $l, r \in T(\Sigma, X)_s$ and $s \in S$, where $\text{Var}(r) \subseteq \text{Var}(l)$. Such rules represent dynamic transitions between states $\langle l \rangle_E$ and $\langle r \rangle_E$. We also allow rules to be labelled and to contain conditions: $[lb] l \Longrightarrow r$ if c , where lb is a (not necessarily unique) label, $c \in T(\Sigma, X)_{\text{bool}}$ and $\text{Var}(c) \subseteq \text{Var}(l)$. Intuitively, the condition means that the rewrite rule can only be applied if term c rewrites to *true*. A *Rewriting*

logic specification is therefore a triple $Spec = (\Sigma, E, R)$ consisting of an algebraic specification (Σ, E) and a set of (conditional) rewrite rules R over Σ and X .

As an example of an RL specification consider a model of a simple dynamic system in which states are multi-sets of symbols A , B , and C . The resulting RL specification $Spec(MS) = (\Sigma, E, R)$ is defined as follows. Let $S = \{ent, ms\}$ be a sort set and let Σ be an S -sorted signature which contains the following function symbols:

$$\begin{array}{ll} A, B, C : ent \in \Sigma, & @ : ent \rightarrow ms \in \Sigma, \\ empty : ms, & @ \otimes @ : ms \ ms \rightarrow ms \in \Sigma, \end{array}$$

(where $@$ is used to indicate the position of an argument in a function symbol to allow for an infix notation). Note that the signature contains an implicit type coercion operator $@ : ent \rightarrow ms$.) The set of equations E contains the equations which axiomatize the associative/commutative properties of a multi-set. Note that the rewrite rules defined below will be applied modulo these equations. Finally, we define R to contain the following three rewrite rules:

$$\begin{array}{ll} [Rule1] A \otimes m1 \Longrightarrow B \otimes m1 & [Rule2] B \otimes C \otimes m1 \Longrightarrow B \otimes A \otimes m1 \\ [Rule3] B \otimes B \otimes m1 \Longrightarrow C \otimes m1. & \end{array}$$

where $m1 \in X_{ms}$. Let $A \otimes C$ be a multi-set representing the initial state of the system. Then the trace $A \otimes C \Longrightarrow B \otimes C \Longrightarrow B \otimes A \Longrightarrow B \otimes B \Longrightarrow C$ represents one possible evolution of the system.

Rewriting Logic provides the notion of a *strategy* for controlling the application of rewrite rules [5,6]. A strategy allows the user to specify the order in which rewrite rules are applied and the possible choices that can be made. The result of applying a strategy is the set of all possible terms that can be produced according to the strategy. A strategy is said to *fail* if it can not be applied (i.e. produces no results). The following is a brief overview of some *elementary strategies* (based on [5,6]):

- (i) **Basic strategy:** lb Any label used in a labelled rule $[lb] \ t \Rightarrow t'$ is a strategy. The result of applying a basic strategy l is the set of all terms that could result from one application of any rule labelled lb .
- (ii) **Concatenation strategy:** $s_1; s_2$ Allows strategies to be sequentially composed, i.e. s_2 is applied to the set of results from s_1 .
- (iii) **Don't know strategy:** $dk(s_1, \dots, s_n)$ Returns the union of all the sets of terms that result from applying each strategy s_1, \dots, s_n .
- (iv) **Don't care strategy:** $dc(s_1, \dots, s_n)$ Chooses nondeterministically to apply one of the strategies s_i which does not fail. The strategy $dc\ one(s_1, \dots, s_n)$ works in a similar way but chooses a *single* result term to return, where as $first(s_1, \dots, s_n)$ applies the first successful strategy in the sequence s_1, \dots, s_n .
- (v) **Iterative strategies:** $repeat^*(s)$ Repeatedly applies s , zero or more times, until the s fails. It returns the last set of results produced before s failed.

As an example, $repeat^*(first(Rule1, Rule2, Rule3))$ is a strategy for $Spec(MS)$ which prioritises the rules so that $Rule1$ is always applied first if it can be, $Rule2$ is applied only if the first rule cannot be applied and $Rule3$ is applied only if the previous two rules cannot be applied.

The above elementary strategy language can be extended to a *defined strategy language* [5,6] which allows recursive strategies to be defined. As an example, consider the simple recursive search strategy $search(i)$ defined below:

$$\begin{aligned} doStep &\Longrightarrow dk(Rule1, Rule2, Rule3) \\ search(i) &\Longrightarrow fail \quad \text{if } i \leq 0 \\ search(i) &\Longrightarrow first(found, doStep; search(i-1)) \quad \text{if } i > 0 \end{aligned}$$

The strategy $search(i)$ repeatedly applies the strategy $doStep$ looking for a multi-set term that satisfies the strategy $found$. It fails if the given maximum number of iterations i is reached. So to search for a multi-set term containing $A \otimes B \otimes C$ we would define the strategy $found$ by the following rewrite rule:

$$[found] \quad A \otimes B \otimes C \otimes m1 \Longrightarrow A \otimes B \otimes C \otimes m1$$

A range of tools have been developed for supporting rewriting logic and strategies, including: MAUDE [13]; ELAN [4,6]; STRATEGO [21]; and TOM [2]. In this paper we have chosen to use ELAN to implement our examples given its simple strategy language and the authors' experience with this tool.

4 Modelling TIMO Using Rewriting Logic and Strategies

In this section we develop a semantic model of TIMO using rewriting logic and strategies, and provide a formal argument of correctness.

4.1 Developing an RL Model for TIMO

Given a TIMO specification TM we consider how to develop a corresponding RL model $RL(TM)$ that correctly captures the meaning of TM . Note for simplicity, the parameters used in communication between processes within TM are restricted to a single location parameter.

We begin by modelling the general concept of a process and network in RL. Let S be the set of sorts in $RL(TM)$ containing: nat for time; $Chan$ for channels; $VLoc$, $ALoc$, and Loc for locations; Prs for processes. and $Nets$ for networks. Coping with the parameter passing that occurs in communication requires careful consideration and for this reason the sort Loc is defined as the union of two subsorts: $VLoc$ represents the input location variables; and $ALoc$ represents the actual locations used in TM .

The S -sorted signature $\Sigma^{RL(TM)}$ for $RL(TM)$ contains the following function symbols to capture the syntax for processes given in Table 1:

$$\begin{aligned} stop &: Prs, & S(@) &: Prs \rightarrow Prs, & @ \mid @ &: Prs \ Prs \rightarrow Prs \\ go(@, @) & \text{ then } @ &: nat \ Loc \ Prs \rightarrow Prs \end{aligned}$$

$$\begin{aligned} in(@, @)(@) \text{ then } @ \text{ else } @ : Chan \text{ nat } VLoc \ Prs \ Prs &\rightarrow Prs \\ out(@, @) < @ > \text{ then } @ \text{ else } @ : Chan \text{ nat } Loc \ Prs \ Prs &\rightarrow Prs. \end{aligned}$$

The function symbol $@ \mid @$ is defined equationally to be associative and commutative as per the definition of TiMO. To model process definitions we add a function symbol $id : s_1 \dots s_n \rightarrow Prs$ for each process identifier $id(u_1, \dots, u_n : s_1, \dots, s_n)$, where s_i is assumed to be a well-defined data type in our model.

We then define the following function symbols to represent networks:

$$@[@] : ALoc \ Prs \rightarrow Nets; \quad @ \mid @ : Nets \ Nets \rightarrow Nets;$$

where $@ \mid @$ is again defined to be associative and commutative.

We now need to formulate appropriate rewrite rules to begin to capture the intended semantics of TiMO. In the RL model developed here we choose the approach of forcing network components with the same location to merge (this turns out to be important since it simplifies the selection of a location to update). The above approach is realized using the rule $al[p1] \mid al[p2] \Rightarrow al[p1 \mid p2]$. Clearly, such a rule is compatible with Eq 3 from Table 2. Each network term will therefore have the form $at_1[pt_1] \mid \dots \mid at_n[pt_n]$, where each location at_i is unique and where each pt_i will represent a set of parallel processes. Any process term which does not contain the parallel operator at its topmost level is referred to as an *atomic process term*. Each individual network location term will have the form $at_i[pt_i^1 \mid \dots \mid pt_i^k]$, where each pt_i^j is an atomic process term.

Next we consider how to model the action rules given in Table 2 within our RL model. First, we define two labelled rules to model the action rule (MOVE):

$$\begin{aligned} [move] \quad al[go(t, al2) \text{ then } p1 \mid p2] &\Longrightarrow al2[S(p1)] \mid al[p2] \\ [move] \quad al[go(t, al2) \text{ then } p1 \mid p2] &\Longrightarrow \\ &al[S(go(t-1, al2) \text{ then } p1) \mid p2] \quad \text{if } t > 0 \end{aligned}$$

The two rules can both be applied when $t > 0$ and this leads to a non-deterministic choice between moving location or allowing time to pass. Note that if $t = 0$ then only the rule that moves to a different location is applicable.

To model the synchronisation required for communication as defined by the action rule (COM) we have the following rule:

$$\begin{aligned} [com] \quad al[out(c, t1) < al1 > \text{ then } p1 \text{ else } p2 \mid in(c, t2)(vl) \text{ then } p3 \text{ else } p4 \mid p5] \\ \Longrightarrow al[S(p1) \mid S(p3[vl/al1]) \mid p5] \end{aligned}$$

This rule makes use of a substitution function $@[@/@] : Prs \ VLoc \ ALoc \rightarrow Prs$, where $pt[vt/at]$ represents the process term that results by substituting all free occurrences (not bound by an input action symbol) of $VLoc$ term vt by the $ALoc$ term at within the process term pt . This function is straightforward to define algebraically using recursion on process terms.

In any TiMO specification TM there will be process definitions of the form $id(u_1, \dots, u_n : s_1, \dots, s_n) \stackrel{\text{df}}{=} P_{id}$ which allow each process identifier $id \in Id$ to be

associated with a well-formed process expression P_{id} (see the action rule (CALL) in Table 2). In $RL(TM)$, for each $id \in Id$ we add a rewrite rule of the form:

$$[calls] \quad al[id(u_1, \dots, u_n) \mid p] \Rightarrow al[RL(P_{id}) \mid p]$$

where $RL(P_{id})$ is the process term that results from translating P_{id} into $RL(TM)$ and each u_i is a variable of sort s_i in $RL(TM)$.

The above labelled rules are collectively referred to as *process transition rules* and are used to define a strategy *step* that represents an update step as follows:

$$step \Longrightarrow repeat*(dc(calls, move, com))$$

The strategy repeatedly applies the three process transition rules and makes use of the *dc* built-in strategy as the order the rules are applied in is irrelevant given that they act on disjoint sets of terms.

In TiMO the last step of any derivation involves applying the (TIME) action rule which allows time to progress and removes all stall symbols. We model this by using a function $tick(@) : Prs \rightarrow Prs$ which is applied to the terms resulting from *step*. We define *tick* recursively as illustrated by the sample rules below:

$$\begin{aligned} tick(stop) &\Longrightarrow stop \\ tick(S(p1)) &\Longrightarrow p1 \\ tick(p1 \mid p2) &\Longrightarrow tick(p1) \mid tick(p2) \\ tick(id(u_1, \dots, u_n)) &\Longrightarrow id(u_1, \dots, u_n) \\ tick((out(a, t) < l > then p1 else p2)) &\Longrightarrow \\ &\quad (out(a, t-1) < l > then p1 else p2) \quad \text{if } t > 0 \\ tick((out(a, 0) < l > then p1 else p2)) &\Longrightarrow p2 \end{aligned}$$

To make the application of this function straightforward we overload *tick* so that it can be applied to networks by defining $tick(@) : Nets \rightarrow Nets$ by

$$tick(al[p]) \Longrightarrow al[tick(p)], \quad tick(n1 \mid n2) \Longrightarrow tick(n1) \mid tick(n2)$$

We can now formulate a rewrite rule *oneStep* in $RL(TM)$ using the strategy *step* and function *tick* that models a derivation step in *TM*:

$$\begin{aligned} [oneStep] \quad al[p] \mid n1 &\Longrightarrow n3 \mid n1 \\ &\text{where } n2 := (step) al[p], \quad n3 := () tick(n2) \end{aligned}$$

The pattern $al[p] \mid n1$ is used to match non-deterministically with a collection of network components (due to the associative/commutative property of $@ \mid @$) and so chooses the next location to update.

It is interesting to note that different semantic choices can be considered for TiMO by appropriately updating the *oneStep* strategy. For example, we could straightforwardly consider a synchronous semantics, introduce priorities to locations or add fairness assumptions. This provides further motivation for developing our RL model.

4.2 Correctness of RL Model

Having developed an RL model for T_{IMO} we now validate that it correctly captures the semantics of T_{IMO}. We do this by showing that our model is *sound* (each step in our RL model represents a derivation step in T_{IMO}) and *complete* (every derivation step possible in T_{IMO} is represented in our RL model). In the sequel let TM be a T_{IMO} specification and let $RL(TM)$ be the corresponding RL model as defined in Section 4.1.

Not all the terms of sort Prs in $RL(TM)$ represent valid processes in TM since they may contain the stall symbol S . Another problem can arise with the improper use of location variables, that is terms of sort $VLoc$, since all uses other than those in an input command need to be bound by an outer input command. We formalise what we mean by a valid process term by defining a function VP .

Definition 1. *The function $VP : T(\Sigma^{RL(TM)})_{Prs} \times \mathcal{P}(T(\Sigma^{RL(TM)})_{VLoc}) \rightarrow \mathbf{B}$ is defined recursively over the structure of process terms as follows:*

$$\begin{aligned}
 VP(stop, VS) &= true \\
 VP(S(pt), VS) &= false \\
 VP(id(v_1, \dots, v_n), VS) &= \begin{cases} true & \text{if } v_i \in VS, \text{ for all } v_i \text{ of sort } VLoc \\ false & \text{otherwise} \end{cases} \\
 VP(go(nt, at) \text{ then } pt, VS) &= VP(pt, VS) \\
 VP(go(nt, vt) \text{ then } pt, VS) &= vt \in VS \wedge VP(pt, VS) \\
 VP(in(ct, nt)(vt) \text{ then } pt_1 \text{ else } pt_2, VS) &= VP(pt_1, VS \cup \{vt\}) \wedge VP(pt_2, VS) \\
 VP(out(ct, nt) < vt > \text{ then } pt_1 \text{ else } pt_2, VS) &= \\
 &\quad VP(pt_1, VS) \wedge vt \in VS \wedge VP(pt_2, VS) \\
 VP(out(ct, nt) < at > \text{ then } pt_1 \text{ else } pt_2, VS) &= VP(pt_1, VS) \wedge VP(pt_2, VS) \\
 VP(pt_1 \mid pt_2, VS) &= VP(pt_1, VS) \wedge VP(pt_2, VS).
 \end{aligned}$$

We define $valPrs(TM) = \{pt \mid pt \in T(\Sigma^{RL(TM)})_{Prs} \text{ and } VP(pt, \{\})\}$ to be the set of all valid process terms and define the set $valNet(TM)$ of valid network terms recursively by: (1) $at[pt] \in valNet(TM)$ if $pt \in valPrs(TM)$; and (2) $net_1 \mid net_2 \in valNet(TM)$, if $net_1, net_2 \in valNet(TM)$.

It can be shown that *oneStep* preserves valid network terms.

Theorem 1. *The strategy *oneStep* is well-defined with respect to valid network terms, i.e. for any $net_1 \in valNet(TM)$, if $net_1 \Longrightarrow net_2$ using *oneStep* then $net_2 \in valNet(TM)$.*

Proof. By the definition of *oneStep* it suffices to consider a valid network location term of the form $at[pt_1 \mid \dots \mid pt_n] \in valNet(TM)$, where $n > 0$ and each pt_i is an atomic process term. It can be seen that each process term pt_i is involved in at most one process transition rule application when *oneStep* is applied. This gives use four possible cases to consider for each pt_i . We omit the full proof for brevity and refer the reader to [11]. \square

We can define an interpretation mapping between T_{IMO} terms in TM and terms in the corresponding RL model $RL(TM)$ as follows.

Definition 2. The process term mapping $\sigma_{Prs} : Prs(TM) \rightarrow valPrs(TM)$ is defined recursively by:

$$\begin{aligned} \sigma_{Prs}(\text{stop}) &= \text{stop}, & \sigma_{Prs}(id(v_1, \dots, v_n)) &= id(v_1, \dots, v_n), \\ \sigma_{Prs}(\text{go}^{\Delta t} l \text{ then } P) &= \text{go}(t, l) \text{ then } \sigma_{Prs}(P), \\ \sigma_{Prs}(a^{\Delta t} ? (vl : Loc) \text{ then } P \text{ else } P') &= \\ & \quad in(a, t)(vl) \text{ then } \sigma_{Prs}(P) \text{ else } \sigma_{Prs}(P'), \\ \sigma_{Prs}(a^{\Delta t} ! \langle l \rangle \text{ then } P \text{ else } P') &= \text{out}(a, t) \langle l \rangle \text{ then } \sigma_{Prs}(P) \text{ else } \sigma_{Prs}(P'), \\ \sigma_{Prs}(P | P') &= \sigma_{Prs}(P) | \sigma_{Prs}(P'). \end{aligned}$$

The network term mapping $\sigma_{Net} : Net(TM) \rightarrow valNet(TM)$ is defined using σ_{Prs} by $\sigma_{Net}(l \llbracket P \rrbracket) = l[\sigma_{Prs}(P)]$ and $\sigma_{Net}(N | N') = \sigma_{Net}(N) | \sigma_{Net}(N')$.

It is straightforward to show that σ_{Prs} and σ_{Net} are bijective mappings and thus have inverses. In order to show the correctness of the RL model we need to prove it is *sound* and *complete* with respect to TIMO (see Figure 1).

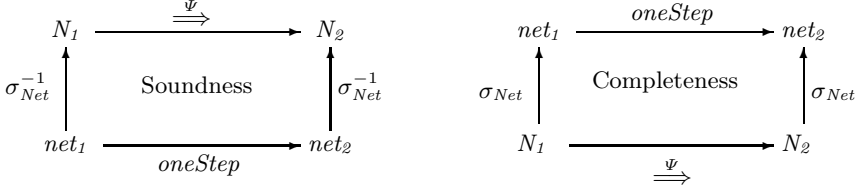


Fig. 1. The properties of soundness and completeness required for $RL(TM)$ to be a correct model of TM

We now show that for any TIMO specification TM the RL model $RL(TM)$ defined in Section 4.1 is a sound and complete model of TM .

Theorem 2 (Soundness). Let $net_1, net_2 \in valNet(TM)$ be any valid network terms. Then if $net_1 \Longrightarrow net_2$ using the strategy $oneStep$ then $\sigma_{Net}^{-1}(net_1) \xRightarrow{\Psi} \sigma_{Net}^{-1}(net_2)$ for some finite multiset $\Psi = \{\psi_1, \dots, \psi_m\}$ of l -actions and some location l . In other words, the diagram for soundness in Figure 1 must commute.

Proof. By the definition of $oneStep$ and the notion of a derivation in TIMO it suffices to consider a valid network location term of the form

$$at[pt_1 | \dots | pt_n] \in valNet(TM),$$

where $n > 0$ and each pt_i is an atomic process term. It can be seen that each process term pt_i is involved in at most one process transition rule application when $oneStep$ is applied. This gives use four possible cases to consider for each pt_i , one for each type of process transition rule and one for when no process transition rules are applicable. The full proof is presented in [11] and for brevity we present only the proof for the $[com]$ rule here.

Suppose the $[com]$ rule is applied to two process terms pt_i and pt_j , ($i \neq j$)

$$\begin{aligned} & out(ct, nt_1) < at_2 > \text{ then } pt_i^1 \text{ else } pt_i^2 \mid in(ct, nt_2)(vt) \text{ then } pt_j^1 \text{ else } pt_j^2 \\ & \Rightarrow S(pt_i^1 \mid S(pt_j^1[vt/at_2])) \end{aligned}$$

The instances of the stall symbol S will be removed by the application of the *tick* function at the end of *oneStep*. Then by the action rule (COM) we have

$$\begin{aligned} & at \llbracket ct^{\Delta nt_1} ! \langle at_2 \rangle \text{ then } \sigma_{Prs}^{-1}(pt_i^1) \text{ else } \sigma_{Prs}^{-1}(pt_i^2) \mid \\ & \quad ct^{\Delta nt_2} ? (vt : Loc) \text{ then } \sigma_{Prs}^{-1}(pt_j^1) \text{ else } pt_j^2 \rrbracket \\ & \xrightarrow{ct < at_2 > @at} at \llbracket \textcircled{S} \sigma_{Prs}^{-1}(pt_i^1) \mid \textcircled{S} \{at_2/vt\} \sigma_{Prs}^{-1}(pt_j^1) \rrbracket \end{aligned}$$

The result follows since the stall symbol \textcircled{S} will be removed by the time progression step in TIMO and since $\sigma_{Prs}^{-1}(pt_j^1[vt/at_2]) = \{at_2/vt\} \sigma_{Prs}^{-1}(pt_j^1)$. \square

Theorem 3 (Completeness). *Let $N_1, N_2 \in Net(TM)$ be any well-formed network terms in TM . Then, if $N_1 \xrightarrow{\Psi} N_2$, for some location l and some multi-set $\Psi = \{\psi_1, \dots, \psi_m\}$ of l -actions, then $\sigma_{Net}(N_1) \Longrightarrow \sigma_{Net}(N_2)$ using *oneStep*. In other words, the diagram for completeness in Figure 7 commutes.*

Proof. By the definition of a derivation in TIMO and the strategy *oneStep* it suffices to consider a well-formed network of the form

$$at \llbracket P_1 \mid \dots \mid P_n \rrbracket \equiv at \llbracket P_1 \rrbracket \mid \dots \mid at \llbracket P_n \rrbracket,$$

where $n > 0$ and each P_i is an atomic process. Suppose $at \llbracket P_1 \mid \dots \mid P_n \rrbracket \xrightarrow{\Psi} N'$, for some finite set of *at*-actions $\Psi = \{\psi_1, \dots, \psi_m\}$, $m \geq 0$. Then it can be seen that each atomic process P_i is involved in at most one *at*-action ψ_i . We show that the derivation applied to each process P_i is correctly captured by the *oneStep* strategy in the RL model. This gives us four possible cases to consider for each P_i : no action rule applied; a (CALL) action rule is applied; a (MOVE) action rule is applied; or a (COM) action rule is applied. We omit the full proof for brevity (see [11]) and present only the proof for the (COM) action rule.

Suppose (COM) has been applied to two processes P_i and P_j , for $i \neq j$, i.e.

$$\begin{aligned} & at \llbracket ct^{\Delta nt_1} ! \langle at_2 \rangle \text{ then } P_i^1 \text{ else } P_i^2 \mid ct^{\Delta nt_2} ? (vt : Loc) \text{ then } P_j^1 \text{ else } P_j^2 \rrbracket \\ & \xrightarrow{ct < at_2 > @at} at \llbracket \textcircled{S} P_i^1 \mid \textcircled{S} \{at_2/vt\} P_j^1 \rrbracket \end{aligned}$$

where the stall symbols \textcircled{S} are removed by the final time step. Then we have

$$\begin{aligned} & \sigma_{Prs}(ct^{\Delta nt_1} ! \langle at_2 \rangle \text{ then } P_i^1 \text{ else } P_i^2 \mid ct^{\Delta nt_2} ? (vt : Loc) \text{ then } P_j^1 \text{ else } P_j^2) \\ & = out(ct, nt_1) < at_2 > \text{ then } \sigma_{Prs}(P_i^1) \text{ else } \sigma_{Prs}(P_i^2) \mid \\ & \quad in(ct, nt_2)(vt) \text{ then } \sigma_{Prs}(P_j^1) \text{ else } \sigma_{Prs}(P_j^2) \end{aligned}$$

By applying the $[calls]$ rule we have

$$\begin{aligned} & out(ct, nt_1) < at_2 > \text{ then } \sigma_{Prs}(P_i^1) \text{ else } \sigma_{Prs}(P_i^2) \mid in(ct, nt_2)(vt) \text{ then } \\ & \quad \sigma_{Prs}(P_j^1) \text{ else } \sigma_{Prs}(P_j^2) \Rightarrow \sigma_{Prs}(P_i^1) \mid \sigma_{Prs}(P_j^1)[vt/at_2] \end{aligned}$$

where all occurrences of the stall symbol S will be removed by the *tick* function. It is then straightforward to see that

$$\sigma_{Prs}(P_i^1 \mid \{at_2/vt\}P_j^1) = \sigma_{Prs}(P_i^1) \mid \sigma_{Prs}(P_j^1)[vt/at_2]$$

by definition of σ_{Prs} and since $\sigma_{Prs}(\{at_2/vt\}P_j^1) = \sigma_{Prs}(P_j^1)[vt/at_2]$. \square

5 An Illustrative Example

In this section we investigate using ELAN [46], a strategy-based rewrite system, to implement a TIMO specification based on our RL model. We consider a small example which provides useful insight into the RL modelling approach used and illustrates the type of (behavioural) properties that can be analysed.

Recall the simple TIMO workflow example introduced in Section 2. The specification WF can be mapped into an RL model $RL(WF)$ as described in Section 4.1 and then investigated using ELAN to provide insight into the behaviour of the original TIMO specification. A range of (behavioural) properties can be analysed including time constraints, use of locations, and causality between actions. For example, consider the following initial TIMO network:

$$Init \llbracket job \mid serv(Web) \rrbracket \mid Web \llbracket serv(Done) \rrbracket$$

After translating this into $RL(WF)$ we can use ELAN to derive the following rewriting trace which shows how a processing job can reach the *Done* location:

$$\begin{aligned} &Init[job \mid serv(Web)] \mid Web[serv(Done)] \Longrightarrow \\ &Init[in(a, 1)(WL) \text{ then } go(1, WL) \text{ then } job \text{ else } job \mid out(a, 2) < Web > \text{ then } \\ &serv(Web) \text{ else } serv(Web)] \mid Web[serv(Done)] \Longrightarrow \\ &Init[go(1, Web) \text{ then } job \mid serv(Web)] \mid Web[serv(Done)] \Longrightarrow \\ &Init[out(a, 2) < Web > \text{ then } serv(Web) \text{ else } serv(Web)] \mid Web[job \mid serv(Done)] \\ &\Longrightarrow \\ &Init[out(a, 2) < Web > \text{ then } serv(Web) \text{ else } serv(Web)] \mid Web[in(a, 1)(WL) \text{ then } \\ &go(1, WL) \text{ then } job \text{ else } job \mid out(a, 2) < Done > \text{ then } serv(Done) \\ &\text{ else } serv(Done)] \Longrightarrow \\ &Init[out(a, 2) < Web > \text{ then } serv(Web) \text{ else } serv(Web)] \mid \\ &Web[go(1, Done) \text{ then } job \mid serv(Done)] \Longrightarrow \\ &Init[out(a, 2) < Web > \text{ then } serv(Web) \text{ else } serv(Web)] \mid \\ &Web[out(a, 2) < Done > \text{ then } serv(Done) \text{ else } serv(Done)] \mid Done[job] \end{aligned}$$

The example trace contains six derivation steps and indeed it is easy to verify using ELAN that this is the smallest number of steps needed in order for a processing job starting at *Init* to reach the *Done* location. Next we consider what happens if we change our network so that it contains a faulty service process:

$$Init \llbracket job \mid serv(Web) \rrbracket \mid Web \llbracket servErr(Done) \rrbracket$$

Again, using ELAN and a simple search strategy we are able to confirm that that it is still possible for a processing job to reach the *Done* location. Furthermore, we can show that it is now possible for a processing job to end up in the *Err* location as the following term derived using ELAN shows:

$$\text{Init}[out(a, 2) < Web > \text{ then } serv(Web) \text{ else } serv(Web)] \mid \\ \text{Web}[out(a, 2) < Err > \text{ then } servErr(Err) \text{ else } servErr(Err)] \mid \text{Err}[job]$$

6 Conclusions

In this paper we have considered using RL to develop a model and implementation of TiMO. The RL model was based on developing a strategy which can capture a maximal parallel computational step of a TiMO specification, including its time rule based previously on negative premises. We have also formally shown the correctness of the resulting semantics by proving it is both sound and complete. We illustrated how the ELAN tool and, in particular, its user defined strategies can be used to model and analyse a TiMO specification. While the example used is intentionally simple for brevity, it still provides an interesting first insight into the range of properties that can be investigated.

TiMO [8] is an appealing process algebra proposed for prototyping software engineering applications where time and mobility are combined. Related models can be found in the literature, such as the timed π -calculus [3], timed distributed π -calculus [12], and timed mobile ambients [1]. RL provides an ideal logical framework for modelling concurrent systems and has been used to model a range of process algebras, such as CCS [16]. In particular, [20] provides a high-level discussion of the use of ELAN for prototyping Π -calculus specifications but while the use of strategies is mentioned no specific details are provided. The RL model of TiMO presented here appears to be novel in its use of strategies to cope with maximal parallel computational steps.

In future work we intend to investigate extending our approach to handle security related aspects of software engineering designs, such as the access permissions defined for TiMO specifications in [10]. Interestingly, the RL model allows a range of semantic choices for TiMO to be considered by changing the derivation step strategy (e.g. adding priorities or fairness assumptions) and we are currently investigating these different semantic choices. We also intend to perform a variety of verification case studies to illustrate the practical application of our methods and investigate its limitations. Finally, we note that at present the analysis of TiMO specifications is limited by the search capabilities and efficiency of ELAN. Work is now underway to develop MAUDE [13] and TOM [2] implementations of the RL model presented here with the aim of improving both the range and efficiency of model analysis.

Acknowledgements. We gratefully acknowledge the financial support of the School of Computing Science, Newcastle University. The work of the first author is supported by CNCS-UEFISCDI project PN-II-ID-PCE-2011-3-0919. We would also like to thank the anonymous referees for their helpful comments and suggestions.

References

1. Aman, B., Ciobanu, G.: Mobile Ambients with Timers and Types. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) ICTAC 2007. LNCS, vol. 4711, pp. 50–63. Springer, Heidelberg (2007)
2. Bolland, E., Brauner, P., Kopetz, R., Moreau, P.-E., Reilles, A.: Tom: Piggybacking Rewriting on Java. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 36–47. Springer, Heidelberg (2007)
3. Berger, M.: Basic Theory of Reduction Congruence for Two Timed Asynchronous p -Calculi. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 115–130. Springer, Heidelberg (2004)
4. Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.-E., Ringeissen, C.: An overview of ELAN. In: Kirchner, C., Kirchner, H. (eds.) WRLA 1998 Electronic Notes in Theoretical Computer Science, vol. 15 (1998)
5. Borovanský, P., Kirchner, C., Kirchner, H., Ringeissen, C.: Rewriting with Strategies in ELAN: A Functional Semantics. *International Journal of Foundations of Computer Science* 12(1), 69–95 (2001)
6. Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.-E.: Elan from a rewriting logic point of view. *Theoretical Computer Science* 285(2), 155–185 (2002)
7. Cardelli, L., Gordon, A.: Mobile Ambients. *Theoretical Computer Science* 240, 170–213 (2000)
8. Ciobanu, G., Koutny, M.: Modelling and Verification of Timed Interaction and Migration. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 215–229. Springer, Heidelberg (2008)
9. Ciobanu, G., Koutny, M.: Timed Mobility in Process Algebra and Petri Nets. *Journal of Algebraic and Logic Programming* 80(7), 377–391 (2011)
10. Ciobanu, G., Koutny, M.: Timed Migration and Interaction with Access Permissions. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 293–307. Springer, Heidelberg (2011)
11. Ciobanu, G., Koutny, M., Steggle, L.J.: A Timed Mobility Semantics based on Rewriting Strategies. TR-1341. School of Computing Science, Newcastle University (2012)
12. Ciobanu, G., Prisacariu, C.: Timers for Distributed Systems. *Electronic Notes in Theoretical Computer Science* 164, 81–99 (2006)
13. Clavel, M., et al.: Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science* 285(2), 187–243 (2002)
14. Corradini, F.: Absolute Versus Relative Time in Process Algebras. *Information and Computation* 156, 122–172 (2000)
15. Hennessy, M.: *A Distributed π -Calculus*. Cambridge University Press (2007)
16. Martí-Oliet, N., Meseguer, J.: Rewriting logic as a logical and semantic framework. *Electronic Notes in Theoretical Computer Science* 4, 190–225 (1996)
17. Meinke, K., Tucker, J.V.: Universal Algebra. In: Abramsky, S., Gabbay, D., Maibaum, T. (eds.) *Handbook of Logic in Computer Science*, vol. I: Mathematical Structures, pp. 189–411, Oxford University (1992)
18. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science* 96(2), 73–155 (1992)
19. Milner, R.: *Communicating and Mobile Systems: the π -Calculus*. Cambridge University Press (1999)
20. Viry, P.: A rewriting implementation of pi-calculus. Technical Report TR-96-30. Dipartimento di Informatica, Università di Pisa, 26 (1996)
21. Visser, E.: Stratego: A Language for Program Transformation Based on Rewriting Strategies. In: Middeldorp, A. (ed.) RTA 2001. LNCS, vol. 2051, pp. 357–361. Springer, Heidelberg (2001)

Towards a Formal Component Model for the Cloud ^{*}

Roberto Di Cosmo¹, Stefano Zacchiroli¹, and Gianluigi Zavattaro²

¹ Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126, CNRS, F-75205 Paris, France
roberto@dicosmo.org,

zack@pps.univ-paris-diderot.fr

² Focus Team, Univ of Bologna/INRIA, Italy
zavattar@cs.unibo.it

Abstract. We consider the problem of deploying and (re)configuring resources in a “cloud” setting, where interconnected software components and services can be deployed on clusters of heterogeneous (virtual) machines that can be created and connected on-the-fly. We introduce the Aeolus component model to capture similar scenarios from realistic cloud deployments, and instrument automated planning of day-to-day activities such as software upgrade planning, service deployment, elastic scaling, etc. We formalize the model and characterize the feasibility and complexity of configuration achievability in Aeolus.

1 Introduction

The expression “*cloud computing*” is broadly used to refer to the possibility of building sophisticated distributed software systems that can be run, on-demand, on a virtualized infrastructure at a fraction of the cost which was necessary just a few years ago. Reaping all the benefits of cloud computing is not easy: while the infrastructure cost falls dramatically, writing a distributed software system that adapts to the demand is difficult, and maintaining and reconfiguring it is a serious challenge.

Several recent industry initiatives strive to address this challenge. CloudFoundry [6] provides tools that allow to select, connect, and push to a cloud well defined services (databases, message buses, ...), that are used as building blocks for writing applications using one of the supported frameworks. Canonical is developing Juju [8], that shares several of CloudFoundry concepts. In the academic world, the Fractal component model [4] focuses on expressivity and flexibility: it provides a general notion of component assembly that can be used to describe concisely, and independently of the programming language, a complex software system. Building on Fractal, FraSCAti [16] provides a middleware that can be used to deploy applications in the cloud.

In all these approaches, the goal is to allow the user to assemble a working system out of components that have been specifically designed or adapted to work together. Component selection and interconnection are the responsibility of the user, and if some reconfiguration needs to happen, it is either obtained by reassembling the system manually, or by writing specific code that is still the responsibility of the user.

* Work partially supported by Aeolus project, ANR-2010-SEGI-013-01, and performed at IRILL, center for Free Software Research and Innovation in Paris, France, <http://www.irill.org>

While *expressivity* is certainly important, solving the cloud challenge also requires *automation*: when the number of components grows, or the need to reconfigure appears more frequently, it is essential to be able to specify at a certain level of abstraction a particular configuration of the distributed software system, and to develop tools that provide a set of possible evolution paths leading from the current system configuration to one that corresponds to a user request.

Automated approaches have been developed for the particular case of configuring *package-based* FOSS (Free and Open Source Software) distributions on a *single* system, and there are generic, solver-based component managers for this task [1].

The goal of this paper is to lay the foundations of such an automated approach for the much more complex situation that arises when one needs to: (re)configure not a single machine, but a variety of possibly “elastic” clusters of heterogeneous machines, living in different domains and offering interconnected services that need to be stopped, modified, and restarted in a specific order for the reconfiguration to be successful.

To this end, we propose (in Sections 2 and 3) a novel component model, called *Aeolus* and loosely inspired by Fractal, where components describe resources which provide and require different functionalities, and may be created or destroyed. As a major difference, though, *Aeolus* components are equipped with *state machines* that describe *declaratively* how required and provided functionalities are enacted. That way we can see *Aeolus* as an *abstraction* of Fractal, yet expressive enough to capture many common deployment scenarii in the cloud. The declarative information is essential to provide a planner with the input needed for exploring the possible evolution paths of the system, and propose a reconfiguration plan, which is the key automation enabler.

In Section 4 we study formally the complexity of finding a deployment plan in *Aeolus*, a property which we call *achievability*. We show that *achievability* is *decidable in polynomial time* if no capacity restriction is imposed on the provided and required functionalities. This simplified model, called *Aeolus⁻*, corresponds to what current mainstream tools can handle, and our result explains why it is still possible, in simple cases, to manage such systems manually.

We show that *achievability becomes undecidable* as soon as one allows to impose restrictions on the number of connections between required and provided functionalities. This limiting result is particularly significant, as some industrial tools are starting to incorporate such restrictions to account for capacity limitations of services in the cloud. The model that we propose to deal with these aspects is called *Aeolus flat* to stress that we do not deal yet with hierarchically nested components or location boundaries, that we will address in future work on a comprehensive *Aeolus* model (Section 6).

2 Use Cases

We introduce the key features of *Aeolus* by eliciting them, step-by-step, from the analysis of realistic scenarii. As a running example, we consider several deployment use cases for WordPress, a popular weblog solution that requires several software services to operate, the main ones being a Web server and a SQL database. We present the use cases in order of increasing complexity ranging from the simplest ones, where everything runs on a single physical machine, to more complex ones where the whole appliance runs on a cloud.

```

Package: wordpress
Version: 3.0.5+dfsg-0+squeeze1
Depends: httpd, mysql-client, php5, php5-mysql, libphp-phpmailer (>= 1.73-4), [...]

Package: mysql-server-5.5
Source: mysql-5.5
Version: 5.5.17-4
Provides: mysql-server, virtual-mysql-server
Depends: libc6 (>= 2.12), zlib1g (>= 1:1.1.4), debconf, [...]
Pre-Depends: mysql-common (>= 5.5.17-4), adduser (>= 3.40), debconf

Package: apache2
Version: 2.4.1-2
Maintainer: Debian Apache Maintainers <debian-apache@...>
Depends: lsb-base, procsps, perl, mime-support, apache2-bin (= 2.4.1-2),
  apache2-data (= 2.4.1-2)
Conflicts: apache2.2-common
Provides: httpd
Description: Apache HTTP Server

```

Fig. 1. Debian package metadata for WordPress, Mysql and the Apache web server (excerpt)

Use case 1 — Package installation. Before considering the services that a machine is offering to others (locally or over the network), we need to model the *software installation* on the machine itself, so we will see how to model the three main components needed by WordPress, as far as their installation is concerned.

Software is often distributed according to the *package* paradigm [7], popularized by FOSS distributions, where software is shipped at the granularity of bundles called *packages*. Each package contains the actual software artifact, its default configuration, as well as a bunch of package metadata.

On a given machine, a software package may exist in different states (e.g. installed or uninstalled) and it should go through a complex sequence of states in different phases of unpacking and configuration to get there. In each of its states, similarly to what happens in most software component models [9], a package may have context *requirements* and offer some features, that we call *provides*. For instance in Debian, a popular FOSS distribution, requirements come in two flavors: *Depends* which must be satisfied before a package can be used, and *Pre-Depends* which must be satisfied before a package can be installed. This distinction is of general interest, as we will see later, so we will distinguish between *weak* requirements and *strong* requirements.

An excerpt of the concrete description of the packages present in Debian for WordPress, Apache2 and MySQL are shown in Fig. 1.

To model a software package at this level of abstraction, we may use a simple state machine, with requirements and provides associated to each state. The ingredients of this model are very simple: a set of states Q , an initial state q_0 , a transition function T from states to states, a set R of requirements, a set P of provides, and a function that maps states to the requirements and provides that are *active* at that state, and tells whether requirements are weak or strong. We call *resource type* any such tuple $\langle Q, q_0, T, P, D \rangle$, which will be formalized in Definition 1.

A system *configuration* built out of a collection of resource types is given by an instance of each resource type, with its current state, and a set of connections between

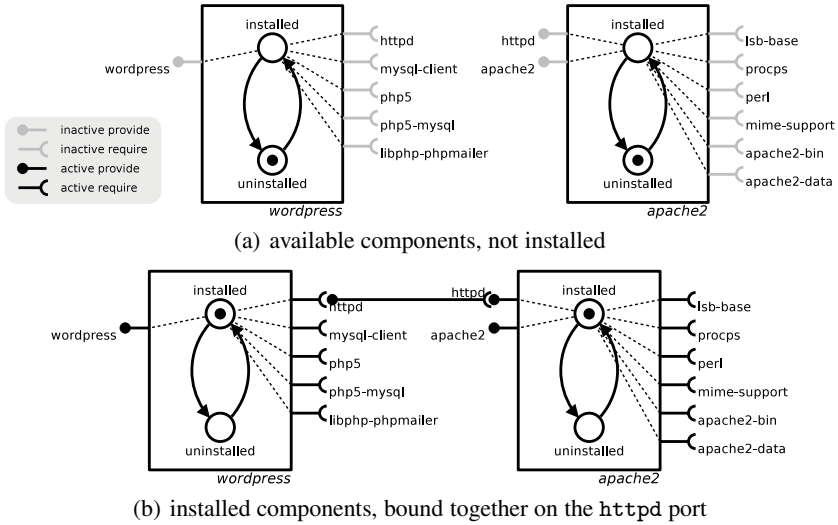


Fig. 2. A simple graphical description of the basic model of a package

requirements and provides of the different resources, that indicate which provide is fulfilling the need of each requirement. A configuration is *correct* if all the requires which are active are satisfied by active provides; this will be made precise in Definition 3.

A natural graphical notation captures all these pieces of information: Fig. 2 presents two correct configurations of a system built using the components from Fig. 1 (only modeling the dependency on `httpd` underlined in the metadata). In Fig. 2(b) the WordPress package is in the installed state, and activates the requirement on `httpd`; Apache2 is also in the installed state, so the `httpd` provide is active and is used to satisfy the requirement, fact which is visualized by the *binding* connecting the two ports.

Use case 2 — Services and packages. Installing the software on a single machine is a process that can already be automated using *package managers*: on Debian for instance, you only need to have an installed Apache server to be able to install WordPress. But bringing it *in production* requires to activate the associated service, which is more tricky and less automated: the system administrator will need to edit configuration files so that WordPress knows the network addresses of an accessible MySQL instance.

The ingredients we have seen up to now in our model are sufficient to capture the dependencies among services, as shown in Fig. 3. There we have added to each package an extra state corresponding to the activation of the associated service, and the strong requirement (graphically indicated by the double tip on the arrow) on `mysql_up` captures the fact that WordPress cannot be started before MySQL is running. In this case, the bindings really correspond to a piece of configuration information, i.e. where to find a suitable MySQL instance.

Notice how this model does not impose any particular way of modeling the relations between packages and services: instead of using a single resource with an installed and

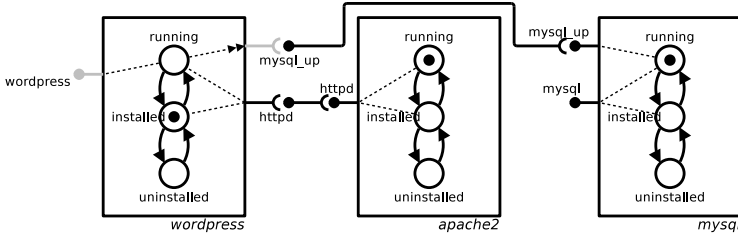


Fig. 3. A graphical description of the basic model of services and packages

a running state, we can also model services and packages as different resources, and relate them through dependencies.

Use case 3 — Redundancy, capacity planning, and conflicts. Services often need to be deployed on different machines to reduce the risk of failure or due to the limitations on the load they can bear. For example, system administrators might want to indicate that a MySQL instance can only support a certain number of WordPress instances. Symmetrically, a WordPress hosting service may want to expose a reverse web proxy / load balancer to the public and require to have a minimum number of *distinct* instances of WordPress available as its back-ends.

To model this kind of situations, we allow capacity information to be added on provides and requires of each resource in Aeolus: a number n on a provide port indicates that it can fulfill no more than n requirements, while a number n on a require port means that it needs to be connected to at least n provides from n *different* components. This information may then be used by a planner to find an optimal replication of the resources to satisfy a user requirement.

As an example, Fig. 4 shows the modeling of a WordPress hosting scenario where we want to offer high availability hosting by putting the Varnish reverse proxy / load balancer in front of several WordPress instances, all connected to a shared replicated MySQL database¹. For a configuration to be correct, the model requires that Varnish is connected to at least 3 (active and distinct) WordPress back-ends, and that each MySQL instance does not serve more than 2 clients.

As a particular case, a 0 constraint on a require means that no provide with the same name can be active at the same time; this can be effectively used to model conflicts between components. For instance, we can use this to model the conflict between the `apache2` and `apache2.2`-common packages that has been omitted in Fig. 2.

Use case 4 — Creating and destroying resources. Use cases like WordPress hosting are commonplace in the cloud, to the point that they are often used to showcase the capabilities of state of the art cloud deployment technologies. The features of the model presented up to here are already expressive enough to encode these *static* deployment scenarios. If one takes Juju’s (rather limiting) assumption that each service is hosted on a separate *machine*, Fig. 4 may then be the representation of the current set of virtual machines (VMs) that we have rented from a public cloud such as Amazon EC2.

¹ All WordPress instances run within separate Apache-s, which have been omitted for simplicity.

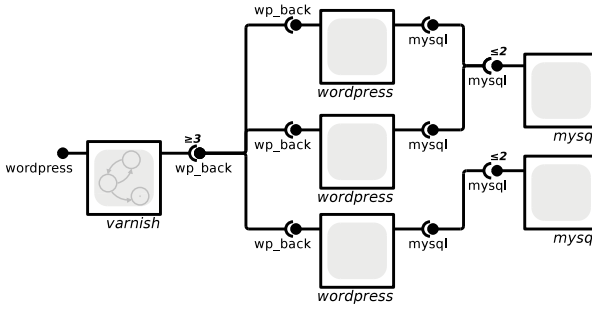


Fig. 4. A graphical description of the model with redundancy and capacity constraints (internal state machines and activation arcs omitted for simplicity)

To model faithfully deployment runs on the cloud, where an arbitrary number of instances of virtual machine images can be allocated and deallocated on the fly, we also allow in our model creation and destruction of all kinds of resources, provided they belong to some existing resource type. This allows to compute reconfiguration plans that create new resources to avoid violating capacity constraints. For instance, in the configuration of Fig. 4 to respond to an increase in traffic load one will need to spawn 2 new WordPress instances, which in turn will require to create new MySQL instances, as the available MySQL-s are not enough to handle the load increase.

3 The Aeolus flat Model

We now formalize the *Aeolus flat model*, that contains all the features elicited from the use cases of the previous section. It is “flat” in the sense that all components live in a single “global” context, are mutually visible, and can connect to each other as long as their ports are compatible.

Notation. We consider the following disjoint sets: \mathcal{I} for interfaces and \mathcal{R} for resources. We use \mathbb{N} to denote strictly positive natural numbers, \mathbb{N}_∞ for \mathbb{N} plus infinity, and \mathbb{N}_0 for \mathbb{N} plus 0.

We model components as finite state automata indicating the current state and possible transitions. When a component changes its state, it can also change the ports that it requires from and provides to other components.

Definition 1 (Resource type). The set \mathcal{T}_{flat} of resource types of the Aeolus flat model, ranged over by $\mathcal{T}_1, \mathcal{T}_2, \dots$ contains 5-ple $\langle Q, q_0, T, P, D \rangle$ where:

- Q is a finite set of states;
- $q_0 \in Q$ is the initial state and $T \subseteq Q \times Q$ is the set of transitions;
- $P = \langle \mathbf{P}, \mathbf{R} \rangle$, with $\mathbf{P}, \mathbf{R} \subseteq \mathcal{I}$, is a pair composed of the set of provide and the set of require ports, respectively;
- D is a function from Q to 3-ple in $(\mathbf{P} \mapsto \mathbb{N}_\infty) \times (\mathbf{R} \mapsto \mathbb{N}_0) \times (\mathbf{R} \mapsto \mathbb{N}_0)$.

Given a state $q \in Q$, the three partial functions in $D(q)$ indicates respectively the provide, weak require, and strong require ports that q activates. The functions associate to the active ports a numerical constraint indicating:

- for provide ports, the maximum number of bindings the port can satisfy,
- for require ports, the minimum number of required bindings to distinct resources,
 - if the number is 0, that indicates a conflict, meaning that there should be no other active port with the same name.

We assume as default constraints ∞ for provide ports (i.e. they can satisfy an unlimited amount of requires) and 1 for require (i.e. one provide is enough to satisfy the requirement). We also assume that the initial state q_0 has no strong demands (i.e. the third function of $D(q_0)$ is empty).

We now define configurations that describe systems composed by components and their bindings. A configuration, ranged over by $\mathcal{C}_1, \mathcal{C}_2, \dots$, is given by a set of resource types, a set of deployed resources in some state, and a set of bindings. Formally:

Definition 2 (Configuration). A configuration \mathcal{C} is a 4-ple $\langle U, Z, S, B \rangle$ where:

- $U \subseteq \mathcal{T}_{flat}$ is the universe of the available resource types;
- $Z \subseteq \mathcal{Z}$ is the set of the currently deployed resources;
- S is the resource state description, i.e. a function that associates to resources in Z a pair $\langle \mathcal{T}, q \rangle$ where $\mathcal{T} \in U$ is a resource type $\langle Q, q_0, T, \mathbf{P}, \mathbf{R}, D \rangle$, and $q \in Q$ is the current resource state;
- $B \subseteq \mathcal{I} \times Z \times Z$ is the set of bindings, namely 3-ple composed by an interface, the resource that requires that interface, and the resource that provides it; we assume that the two resources are distinct.

Notation. We write $\mathcal{C}[z]$ as a lookup operation that retrieves the pair $\langle \mathcal{T}, q \rangle = S(z)$, where $\mathcal{C} = \langle U, Z, S, B \rangle$. On such a pair we then use the postfix projection operators .type and .state to retrieve \mathcal{T} and q , respectively. Similarly, given a resource type $\langle Q, q_0, T, \langle \mathbf{P}, \mathbf{R} \rangle, D \rangle$, we use projections to (recursively) decompose it: .states , .init , and .trans return the first three elements; .prov , .req return \mathbf{P} and \mathbf{R} ; $\mathbf{P}\text{map}(q)$, $\mathbf{R}_w\text{map}(q)$, and $\mathbf{R}_s\text{map}(q)$ return the three elements of the $D(q)$ tuple. When there is no ambiguity we take the liberty to apply the resource type projections to $\langle \mathcal{T}, q \rangle$ pairs. *Example:* $\mathcal{C}[z].\mathbf{R}_s\text{map}(q)$ stands for the strong require ports (and their arities) of resource z in configuration \mathcal{C} when it is in state q .

We are now ready to formalize the notion of configuration correctness. We consider two distinct notions of correctness: *weak* and *strong*. According to the former, only weak requirements are considered, while the latter also considers strong ones. Intuitively, weak correctness can be temporarily violated during the deployment of a new component configuration, but needs to be fulfilled at the end; strong correctness, on the other hand, shall never be violated.

Definition 3 (Correctness). Let us consider the configuration $\mathcal{C} = \langle U, Z, S, B \rangle$.

We write $\mathcal{C} \models_{req} \langle z, r, n \rangle$ to indicate that the require port of resource z , with interface r , and associated number n is satisfied. Formally, if $n = 0$ all resources other than z cannot have an active provide port with interface r , namely for each $z' \in Z \setminus \{z\}$ such that $\mathcal{C}[z'] = \langle \mathcal{T}', q' \rangle$ we have that r is not in the domain of $\mathcal{T}'.\mathbf{P}\text{map}(q')$. If $n > 0$ then the port is bound to at least n active ports, i.e. there exist n distinct resources $z_1, \dots, z_n \in Z \setminus \{z\}$ such that for every $1 \leq i \leq n$ we have that $\langle r, z, z_i \rangle \in B$, $\mathcal{C}[z_i] = \langle \mathcal{T}^i, q^i \rangle$ and r is in the domain of $\mathcal{T}^i.\mathbf{P}\text{map}(q^i)$.

Similarly for provides, we write $\mathcal{C} \models_{\text{prov}} (z, p, n)$ to indicate that the provide port of resource z , with interface p , and associated number n is not bound to more than n active ports. Formally, there exist no m distinct resources $z_1, \dots, z_m \in Z \setminus \{z\}$, with $m > n$, such that for every $1 \leq i \leq m$ we have that $\langle p, z_i, z \rangle \in B$, $S(z_i) = \langle \mathcal{T}^i, q^i \rangle$ and p is in the domain of $\mathcal{T}^i \cdot \mathbf{R}_{w, \text{map}}(q^i)$ or $\mathcal{T}^i \cdot \mathbf{R}_{s, \text{map}}(q^i)$.

The configuration \mathcal{C} is correct if for each resource z in Z , given $S(z) = \langle \mathcal{T}, q \rangle$ with $\mathcal{T} = \langle Q, q_0, T, P, D \rangle$ and $D(q) = \langle \mathcal{P}, \mathcal{R}_w, \mathcal{R}_s \rangle$, we have that $(p \mapsto n_p) \in \mathcal{P}$ implies $\mathcal{C} \models_{\text{prov}} (z, p, n_p)$, and $(r \mapsto n_r) \in \mathcal{R}_w$ implies $\mathcal{C} \models_{\text{req}} (z, r, n_r)$, and $(r \mapsto n'_r) \in \mathcal{R}_s$ implies $\mathcal{C} \models_{\text{req}} (z, r, n'_r)$.

Analogously we say that it is strong correct if only the strong requirements are considered: namely, we require $(p \mapsto n_p) \in \mathcal{P}$ implies $\mathcal{C} \models_{\text{prov}} (z, p, n_p)$ and $(r \mapsto n_r) \in \mathcal{R}_s$ implies $\mathcal{C} \models_{\text{req}} (z, r, n_r)$.

As our main interest is planning, we now formalize how configurations evolve from one state to another, by the means of atomic actions.

Definition 4 (Actions). The set \mathcal{A} contains the following actions:

- $\text{stateChange}(z, q_1, q_2)$ where $z \in \mathcal{Z}$;
- $\text{bind}(r, z_1, z_2)$ where $z_1, z_2 \in \mathcal{Z}$ and $r \in \mathcal{I}$;
- $\text{unbind}(r, z_1, z_2)$ where $z_1, z_2 \in \mathcal{Z}$ and $r \in \mathcal{I}$;
- $\text{newRsrc}(z: \mathcal{T})$ where $z \in \mathcal{Z}$ and \mathcal{T} is a
- $\text{delRsrc}(z)$ where $z \in \mathcal{Z}$.

The execution of actions can now be formalized using a labeled transition systems on configurations, which uses actions as labels.

Definition 5 (Reconfigurations). Reconfigurations are denoted by transitions $\mathcal{C} \xrightarrow{\alpha} \mathcal{C}'$ meaning that the execution of $\alpha \in \mathcal{A}$ on the configuration \mathcal{C} produces a new configuration \mathcal{C}' . The transitions from a configuration $\mathcal{C} = \langle U, Z, S, B \rangle$ are defined as follows:

$$\begin{aligned} \mathcal{C} \xrightarrow{\text{stateChange}(z, q_1, q_2)} \langle U, Z, S', B \rangle & \quad \mathcal{C} \xrightarrow{\text{bind}(r, z_1, z_2)} \langle U, Z, S, B \cup \langle r, z_1, z_2 \rangle \rangle \\ \text{if } \mathcal{C}[z].\text{state} = q_1 & \quad \text{if } \langle r, z_1, z_2 \rangle \notin B \\ \text{and } (q_1, q_2) \in \mathcal{C}[z].\text{trans} & \quad \text{and } r \in \mathcal{C}[z_1].\text{req} \cap \mathcal{C}[z_2].\text{prov} \\ \text{and } S'(z') = \begin{cases} \langle \mathcal{C}[z].\text{type}, q_2 \rangle & \text{if } z' = z \\ \mathcal{C}[z'] & \text{otherwise} \end{cases} & \end{aligned}$$

$$\mathcal{C} \xrightarrow{\text{unbind}(r, z_1, z_2)} \langle U, Z, S, B \setminus \langle r, z_1, z_2 \rangle \rangle \quad \text{if } \langle r, z_1, z_2 \rangle \in B$$

$$\begin{aligned} \mathcal{C} \xrightarrow{\text{newRsrc}(z: \mathcal{T})} \langle U, Z \cup \{z\}, S', B \rangle & \quad \mathcal{C} \xrightarrow{\text{delRsrc}(z)} \langle U, Z \setminus \{z\}, S', B' \rangle \\ \text{if } z \notin Z, \mathcal{T} \in U & \quad \text{if } S'(z') = \begin{cases} \perp & \text{if } z' = z \\ \mathcal{C}[z'] & \text{otherwise} \end{cases} \\ \text{and } S'(z') = \begin{cases} \langle \mathcal{T}, \mathcal{T}.\text{init} \rangle & \text{if } z' = z \\ \mathcal{C}[z'] & \text{otherwise} \end{cases} & \quad \text{and } B' = \{ \langle r, z_1, z_2 \rangle \in B \mid z \notin \{z_1, z_2\} \} \end{aligned}$$

Notice that in the definition of the transitions there is no requirement on the reached configuration: the correctness of these configurations will be considered at the level of a deployment run.

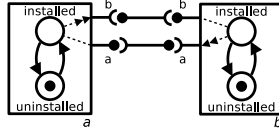


Fig. 5. On the need of a *multiple state change* action: how to install *a* and *b*?

Also, we observe that there are configurations that cannot be reached through sequences of the actions we have introduced. In Fig. 5 for instance, there is no way for package *a* and *b* to reach the installed state, as each package require the other to be installed *first*. In practice, when confronted with such situations—that can be found for example in FOSS distributions in the presence of *Pre-Depend* loops—current tools either perform all the state changes atomically, or abort deployment.

We want our planners to be able to propose reconfigurations containing such atomic transitions, as long as that is the only way to reach a requested configuration. To this end, we introduce the notion of *multiple state change*.

Definition 6 (Multiple state change).

A multiple state change $\mathcal{M} = \{stateChange(z^1, q_1^1, q_2^1), \dots, stateChange(z^l, q_1^l, q_2^l)\}$ is a set of state change actions on different resources (i.e. $z^i \neq z^j$ for every $1 \leq i < j \leq l$). We use $\langle U, Z, S, B \rangle \xrightarrow{\mathcal{M}} \langle U, Z, S', B \rangle$ to denote the effect of the simultaneous execution of the state changes in \mathcal{M} : formally, $\langle U, Z, S, B \rangle \xrightarrow{stateChange(z^1, q_1^1, q_2^1)} \dots \xrightarrow{stateChange(z^l, q_1^l, q_2^l)} \langle U, Z, S', B \rangle$.

Notice that the order of execution of the state change actions does not matter as all the actions are executed on different resources.

We can now define a *deployment run*, which is a sequence of actions that transform an initial configuration into a final correct one without violating strong correctness along the way. A deployment run is the output we expect from a planner, when it is asked how to reach a desired target configuration.

Definition 7 (Deployment run). A deployment run is a sequence $\alpha_1 \dots \alpha_m$ of actions and multiple state changes such that there exist \mathcal{C}_i such that $\mathcal{C} = \mathcal{C}_0, \mathcal{C}_{j-1} \xrightarrow{\alpha_j} \mathcal{C}_j$ for every $j \in \{1, \dots, m\}$, and the following conditions hold:

configuration correctness \mathcal{C}_0 and \mathcal{C}_m are correct while, for every $i \in \{1, \dots, m-1\}$, \mathcal{C}_i is strong correct;

multi state change minimality if α_j is a multiple state change then there exists no proper subset $\mathcal{M} \subset \alpha_j$, or state change action $\alpha \in \alpha_j$, and correct configuration \mathcal{C}' such that $\mathcal{C}_{j-1} \xrightarrow{\mathcal{M}} \mathcal{C}'$, or $\mathcal{C}_{j-1} \xrightarrow{\alpha} \mathcal{C}'$.

We now have all the ingredients to define the notion of *achievability*, that is our main concern: given an universe of resource types, we want to know whether it is possible to deploy at least one resource of a given resource type \mathcal{T} in a given state q .

Definition 8 (Achievability problem). The achievability problem has as input an universe U of resource types, a resource type \mathcal{T} , and a target state q . It returns as output

true if there exists a deployment run $\alpha_1 \dots \alpha_m$ such that $\langle U, \emptyset, \emptyset, \emptyset \rangle \xrightarrow{\alpha_1} \mathcal{C}_1 \xrightarrow{\alpha_2} \dots \xrightarrow{\alpha_m} \mathcal{C}_m$ and $\mathcal{C}_m[z] = \langle \mathcal{T}, q \rangle$, for some resource z in \mathcal{C}_m . Otherwise, it returns **false**.

Notice that the restriction in this decision problem to one resource in a given state is not limiting: one can easily encode any given final configuration by adding a dummy provide port enabled only by the required final states and a dummy component with weak requirements on all such provides.

4 Achievability

In this section, we establish our main results concerning the difficulty of the achievability problem. The results change significantly depending on the restrictions imposed on the numerical constraints that are allowed as co-domains of the three $D(q)$ partial functions. We consider here two extreme cases, which are detailed in the table below:

| model | $co\text{-domain}(\mathbf{Pmap}())$ | $co\text{-domain}(\mathbf{R}_w\text{map}())$ | $co\text{-domain}(\mathbf{R}_s\text{map}())$ |
|----------------|-------------------------------------|--|--|
| $Aeolus^-$ | $\{\infty\}$ | $\{1\}$ | $\{1\}$ |
| $Aeolus\ flat$ | \mathbb{N}_∞ | \mathbb{N}_0 | \mathbb{N}_0 |

$Aeolus\ flat$ is the same model of Def. [II](#) while $Aeolus^-$ is a restriction of it where only the default numerical constraints can be used: provide ports can always serve an unlimited amount of bindings, and require ports cannot conflict with other active ports, nor require a minimum number of bindings strictly higher than 1. In the following we will show that achievability is decidable in $Aeolus^-$, but undecidable in $Aeolus\ flat$.

Achievability Is decidable in $Aeolus^-$. We start by presenting a decision algorithm for the achievability problem in $Aeolus^-$. The idea is to perform an abstract forward exploration of all reachable configurations. Before presenting the algorithm, we list the properties of $Aeolus^-$ we exploit:

- as in $Aeolus^-$ the value 0 on require ports is forbidden, the addition to a configuration of new components cannot forbid the execution of formerly possible actions;
- as in $Aeolus^-$ provide ports have capacity ∞ and require ports have numerical constraint 1, the correctness of a configuration can be checked simply by verifying that the set of active require ports is a subset of the set of active provide ports.

In the light of the second observation, and knowing that the sets of active require and provide ports are functions of the internal state of the components, we abstractly represent configurations simply as sets of pairs $\langle \mathcal{T}, q \rangle$ indicating the type and the state of the components in the configuration. This way, symbolic configurations abstract away from the exact number of instances of each kind of component, and from their current bindings.

We consider symbolic runs representing the evolutions of abstract configurations. Moreover, thanks to the first observation, we can restrict ourselves to consider only evolutions where the set of available pairs $\langle \mathcal{T}, q \rangle$ does not decrease. Namely, we perform a symbolic forward exploration starting from an abstract configuration containing all the

Algorithm 1. Checking achievability in the Aeolus^- model

```

function ACHIEVABILITY(universe of resources  $U$ , resource type  $\mathcal{T}$ , state  $q$ )
   $absConf := \{\langle \mathcal{T}', \mathcal{T}'.init \rangle \mid \mathcal{T}' \in U\}$ 
   $provPort := \bigcup_{\langle \mathcal{T}', q' \rangle \in absConf} \{dom(\mathcal{T}'.Pmap(q'))\}$ 
  repeat
     $new := \{\langle \mathcal{T}', q' \rangle \mid \langle \mathcal{T}', q'' \rangle \in absConf, (q'', q') \in \mathcal{T}'.trans\} \setminus absConf$ 
     $newPort := \bigoplus_{\langle \mathcal{T}', q' \rangle \in new} \{dom(\mathcal{T}'.Pmap(q'))\}$ 
    while  $\exists \langle \mathcal{T}', q' \rangle \in new$  s.t.  $dom(\mathcal{T}'.Rmap(q')) \not\subseteq provPort \cup newPort$  do
       $new := new \setminus \{\langle \mathcal{T}', q' \rangle\}$ 
       $newPort := newPort \ominus \{dom(\mathcal{T}'.Pmap(q'))\}$ 
    end while
     $absConf := absConf \cup new$ 
     $provPort := provPort \cup newPort$ 
  until  $new = \emptyset$ 
  if  $\langle \mathcal{T}, q \rangle \in absConf$  and  $dom(\mathcal{T}.Rwmap(q)) \subseteq provPort$  then return true
  else return false
  end if
end function

```

pairs $\langle \mathcal{T}', \mathcal{T}'.init \rangle$ representing components in their initial state. Then we extend the abstract configuration by adding step-by-step new pairs $\langle \mathcal{T}', q' \rangle$.

Algorithm 1 checks achievability by relying on two auxiliary data structures: $absConf$ is the set of pairs $\langle \mathcal{T}', q' \rangle$ indicating the type and state of the components in the current abstract configuration, and $provPort$ is the set of provide ports active in such a configuration. The algorithm incrementally extends $absConf$ until it is no longer possible to add new pairs.

At each iteration, the potential new pairs are initially computed by checking the automata transitions, and stored in the set new . Not all those states could be actually reached as one needs to check whether their strongly require ports are included in the available provide ports $provPort$ or in the ports opened by the new states. This is done by a one-by-one elimination of pairs $\langle \mathcal{T}', q' \rangle$ from new when their strong requirements are unsatisfiable. During elimination, we use $newPort$, a *multiset* of the provide ports which are activated by the component states currently in new . We use double curly braces for multisets, and \oplus and \ominus for multiset union and difference.

When the final sets $absConf$ and $provPort$ are computed, achievability for the resource type \mathcal{T} and state q can be simply checked by verifying the presence of $\langle \mathcal{T}, q \rangle$ in $absConf$, and by controlling whether its (weak) requirements are satisfied by the active provide ports $provPort$. Strong requirements are satisfied by construction.

We are now ready to prove our decidability result for the Aeolus^- model.

Theorem 1. *Let U be a set of resource types of the Aeolus^- model. Given the resource type \mathcal{T} and the state q , the achievability problem for U , \mathcal{T} , and q can be checked in polynomial time (with respect to the size of the descriptions of the resources in U).*

Proof. The symbolic representation of the initial configuration $\langle U, \emptyset, \emptyset, \emptyset \rangle$ is included in the initial set $absConf$. It is easy to see that given the transition $\mathcal{C} \xrightarrow{\alpha} \mathcal{C}'$ of a deployment run, if the symbolic representation of \mathcal{C} is included in $absConf$ at the beginning of an iteration of the **repeat**, then the symbolic representation of \mathcal{C}' will be surely included

in *absConf* at the end of such iteration. Therefore, if there exists a deployment run able to achieve a component of type \mathcal{T} in the state q , then the Algorithm [1](#) will detect achievability. This proves that the algorithm is complete.

The soundness of the algorithm follows from the following argument. The symbolic forward exploration performed by the algorithm corresponds to an actual deployment run that initially creates sufficiently many components in order to guarantee that all the state changes considered by the symbolic exploration can be actually executed, and every time an action changes the state of one component of type \mathcal{T}' from q'' to q' , there exists at least another component in the concrete system of type \mathcal{T}' which remains in state q'' .

The polynomial complexity of the algorithm follows from the fact that both the **repeat** and the **while** cycles perform a number of iterations smaller than the number of different pairs $\langle \mathcal{T}', q' \rangle$ in the universe of resource types U . \square

Achievability Is undecidable in Aeolus flat. We now show that the decision procedure for achievability of the previous section cannot be extended to deal with the Aeolus flat model. In fact, for this last model achievability turns out to be undecidable. The proof is by reduction from the reachability problem in 2 Counter Machines (2CMs) [\[11\]](#), a well-known Turing-complete computational model. A 2CM is a machine with *two registers* R_1 and R_2 holding arbitrary large natural numbers and a *program* P consisting of a finite sequence of numbered instructions of the following type:

- $j : \text{Inc}(R_i)$: increments R_i and goes to the instruction $j + 1$;
- $j : \text{DecJump}(R_i, l)$: if the content of R_i is not zero, then decreases it by 1 and goes to the instruction $j + 1$, otherwise jumps to the instruction l .

A state of the machine is given by a tuple (i, v_1, v_2) where i indicates the next instruction to execute (the program counter) and v_1 and v_2 are the contents of the two registers. It is not restrictive to assume that the registers are initially set to zero.

We model a 2CM as follows. We use a component to simulate the execution of the program instructions. The contents v_i of the register R_i is modeled by v_i components in a particular state q_i . Increment instructions add one component in this state q_i , while decrement instructions move one component in state q_i to a different state. The state q_i activates a provide port one_i , so the simulation of a jump has simply to check the absence in the environment of active one_i ports.

The resource types of the components that we use to model 2CMs are depicted in Fig. [6](#). Namely, we consider four resource types: \mathcal{T}_P to simulate the execution of the program instructions, \mathcal{T}_{R_1} and \mathcal{T}_{R_2} for the two registers and \mathcal{T}_B used to guarantee that components of type \mathcal{T}_{R_i} involved in the simulation cannot be deleted. In \mathcal{T}_P we assume one state q_j for each instruction j . If the j -th instruction is $j : \text{Inc}(R_i)$, a protocol with three intermediary states is executed. The first one will activate a port on_i allowing a resource of type \mathcal{T}_{R_i} to start a complementary protocol. The second state of the protocol activates a strong requirement on the provide port inc_i while the last state activates a conflict on the same port inc_i . The complementary protocol of the resource type \mathcal{T}_{R_i} includes three states as well. The first one activates a strong requirement on the port on_i ; in this way, the protocol can start only if the complementary protocol already started. The second state of the protocol activates the port inc_i in order to allow the protocol

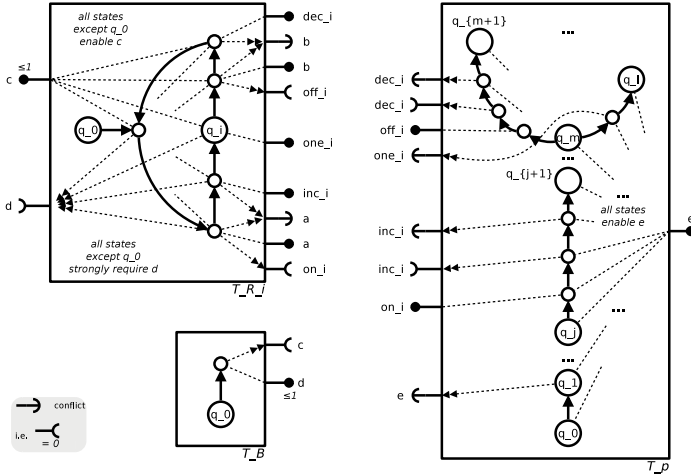


Fig. 6. Modeling 2 counter machines (2CMs) in the Aeolus flat model (sketch)

to progress. Finally, the protocol completes by entering the q_i state. Note that the first state of the protocol opens a provide port a , and the first two states activates a strong requirement on the absence in the environment of such an active port. This guarantees that exactly one resource of type \mathcal{T}_{R_i} will execute the protocol, in other terms, the register R_i is incremented exactly by 1.

Fig. 6 also depicts the modeling of an instruction $m : \text{DecJump}(R_i, l)$. The decrement branch executes a protocol similar to the previous one, whose effect here is to decrement R_i by 1. The jump branch simply checks the absence of components of type \mathcal{T}_{R_i} in state q_i by activating a strong requirement on the absence of an active port one_i (note that such a port is indeed activated by components of type \mathcal{T}_{R_i} in state q_i).

In our component model, when a resource z is not used to satisfy strong requirements, it could be removed by executing the $delRsrc(z)$ action. The cancellation of a components of type \mathcal{T}_{R_i} could then erroneously change register contents during simulation. To avoid that we force the connection of each resource of type \mathcal{T}_{R_i} with a corresponding instance of a component of type \mathcal{T}_B . These types of resources reciprocally “strongly” connect through the ports c and d as soon as they move from their initial state q_0 . Such connections remain active during the entire simulation, ensuring components will not be deleted by mistake. Notice that it is necessary to add the capacity constraint 1 to the provide ports c and d , in order to have an exact one-to-one correspondence between the components of type \mathcal{T}_{R_i} and those of type \mathcal{T}_B .

As a final remark, notice that the first state q_1 of the resource type \mathcal{T}_P has a strong requirement on the absence in the environment of an active provide port e , port which is activated by all the states in \mathcal{T}_P . This guarantees that at most one component of type \mathcal{T}_P will simulate program instructions. Moreover, we also have to avoid that such component is removed by a $delRsrc$ action during the simulation: this can be guaranteed by using the same pairing technique with a component of type \mathcal{T}_B described above. It is sufficient to impose that all the states of \mathcal{T}_P , but q_0 , activate a provide port on c with

numerical constraint 1, and a strongly require port on d . For simplicity, this part of the specification of \mathcal{T}_P is not shown in Fig. 6.

We are now ready to formally state our undecidability result.

Theorem 2. *The achievability problem is undecidable in the Aeolus flat model.*

Proof. Let M be a 2CM and let $U = \{\mathcal{T}_P, \mathcal{T}_{R_1}, \mathcal{T}_{R_2}, \mathcal{T}_B\}$ be the set of the corresponding resource types defined as in Fig. 6. In the light of the discussion above, we have that achievability is satisfied for the universe U , the resource type \mathcal{T}_P and the state q_j if and only if the j -th instruction is reachable in M . The undecidability of achievability thus follows from the undecidability of reachability for 2CMs. \square

5 Related Work

To the best of our knowledge this is the first paper that formally addresses the problem of component deployment in the cloud. In this section we compare the approach we have adopted to related formal models considered in slightly different contexts.

Automata have been adopted long ago in the context of component-oriented development frameworks. One of the most influential model are *interface automata* [3], where automata are used to represent the component behavior in terms of input, output, and internal actions. Interface automata support automatic compatibility check and refinement verification: a component refines another if its interface has weaker input assumptions and stronger output guarantees. Differently from that approach, we are not interested in component compatibility or refinement, and we do not require complementary behavior of components: we simply check in the current configuration whether all required functionalities are provided by currently deployed components. The automata in Aeolus do not represent the internal behavior of components, but the effect on the component of an external deployment or reconfiguration actions.

Aeolus reconfiguration actions show interesting similarities with transitions in Petri nets [13], a very popular model born from the attempt to extend automata with concurrency. At first sight, one might encode our model in Petri net, representing our component states as places, each deployed component as a token in the corresponding place, and reconfiguration actions as transitions that cancel and produce tokens. Achievability in Aeolus would then correspond to *coverability* in Petri nets. But there are several important differences. Multiple state change actions can atomically change the state of an unbounded number of components, while in Petri net each transition consumes a predefined number of tokens. More importantly, we have proved that achievability can be solved in polynomial time for the Aeolus⁻ fragment and that it is undecidable for the Aeolus flat model, while in Petri nets coverability is an ExpSpace problem [14].

Several process calculi extend/modify the π -calculus [10] in order to deal with software components. The Piccola calculus [2] extends the asynchronous π -calculus [10] with *forms*, first-class extensible namespaces, useful to model component interfaces and bindings. Calculi like KELL [15] and HOMER [5] extends a core π -calculus with hierarchical locations, local actions, higher-order communication, programmable membranes, and dynamic binding. More recently, MECo [12] has extended this approach by proposing also explicit component interfaces and channels to realize tunneling effects

traversing the hierarchical location boundaries. On the one hand, all these proposals differ from *Aeolus* model because they focus on the modeling of component interactions and communication, while we focus on their interdependencies during system deployment and reconfiguration. On the other hand, we plan to take inspiration from these calculi in order to extend our model with boundaries and administrative domains.

6 Conclusions and Future Work

We have presented *Aeolus flat*, a component model expressive enough to capture most common deployment scenarii for distributed software applications in the cloud. We have shown that it is possible to generate a deployment plan in polynomial time for the fragment *Aeolus⁻* of the model, corresponding to the industrial tools currently in use, while it is not possible to generate a deployment plan for *Aeolus flat*, that captures more faithfully the constraints imposed by real world applications.

Several interesting models between *Aeolus⁻* and *Aeolus flat* can now be considered, to reconcile expressivity and decidability: one can impose in *Aeolus flat* an upper limit on the number of resources that can be allocated during a deployment run; or one can extend *Aeolus⁻* with only conflict constraints.

The *Aeolus flat* model can be extended to a hierarchical component model to take into account administrative domains and components that are built by grouping together other components. We will also experiment with different planning systems to explore the issues related to plan generation, and use the results as additional guidance in the search for the best compromise between expressivity and decidability.

References

1. Abate, P., Di Cosmo, R., Treinen, R., Zacchiroli, S.: MPM: a modular package manager. In: CBSE 2011: 14th Symposium on Component Based Software Eng., pp. 179–188. ACM (2011)
2. Achermann, F., Nierstrasz, O.: A calculus for reasoning about software composition. *Theor. Comput. Sci.* 331(2-3), 367–396 (2005)
3. de Alfaro, L., Henzinger, T.A.: Interface automata. In: ESEC / SIGSOFT FSE (2001)
4. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The fractal component model and its support in java. *Softw., Pract. Exper.* 36(11-12), 1257–1284 (2006)
5. Bundgaard, M., Hildebrandt, T.T., Godskenen, J.C.: A cps encoding of name-passing in higher-order mobile embedded resources. *Theor. Comput. Sci.* 356(3), 422–439 (2006)
6. Cloud Foundry, deploy & scale your applications in seconds (retrieved April 2012), <http://www.cloudfoundry.com/>
7. Di Cosmo, R., Trezentos, P., Zacchiroli, S.: Package upgrades in FOSS distributions: Details and challenges. In: HotSWup 2008 (2008)
8. Juju, devops distilled (retrieved April 2012), <https://juju.ubuntu.com/>
9. Lau, K.K., Wang, Z.: Software component models. *IEEE Trans. Software Eng.* 33(10), 709–724 (2007)
10. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes. *i/ii. Inf. Comput.* 100(1), 1–77 (1992)
11. Minsky, M.: *Computation: finite and infinite machines*. Prentice-Hall (1967)

12. Montesi, F., Sangiorgi, D.: A Model of Evolvable Components. In: Wirsing, M., Hofmann, M., Rauschmayer, A. (eds.) TGC 2010, LNCS, vol. 6084, pp. 153–171. Springer, Heidelberg (2010)
13. Petri, C.A.: Kommunikation mit Automaten. PhD thesis. Institut für Instrumentelle Mathematik, Bonn, Germany (1962)
14. Rackoff, C.: The covering and boundedness problems for vector addition systems. *Theoret. Comp. Sci.* 6, 223–231 (1978)
15. Schmitt, A., Stefani, J.B.: The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In: Priami, C., Quaglia, P. (eds.) GC 2004. LNCS, vol. 3267, pp. 146–178. Springer, Heidelberg (2005)
16. Seinturier, L., Merle, P., Fournier, D., Dolet, N., Schiavoni, V., Stefani, J.B.: Reconfigurable SCA applications with the FraSCAti platform. In: IEEE SCC, pp. 268–275. IEEE (2009)

The Rely/Guarantee Approach to Verifying Concurrent BPEL Programs

Huibiao Zhu^{1,*}, Qiwen Xu², Chris Ma³, Shengchao Qin⁴, and Zongyan Qiu⁵

¹ Software Engineering Institute, East China Normal University

² Faculty of Science and Technology, University of Macau

³ Computer Service Centre, Macau Polytechnic Institute

⁴ School of Computing, University of Teesside

⁵ School of Mathematical Sciences, Peking University

hbzhu@sei.ecnu.edu.cn

Abstract. Web services have become more and more important in these years, and BPEL4WS (BPEL) is the OASIS standard for web services composition and orchestration. It contains several distinct features, including scope-based compensation and fault handling mechanism. This paper focuses on the verification of BPEL programs, especially the verification of concurrent BPEL programs. The rely/guarantee approach is applied. Firstly, we present the operational semantics for BPEL programs. Secondly we apply the rely/guarantee method in the design of the verification rules. The rules can handle the features of BPEL programs, including compensation, fault handling and concurrency. Finally, the whole proof system is proved to be sound based on our operational semantics.

1 Introduction

Web services and other web-based applications have been becoming more and more important in practice. In this flowering field, various web-based business process languages have been introduced, such as XLANG [21], WSFL [15], BPEL4WS (BPEL) [8] and StAC [4], which are designed for the description of services composed of a set of processes across the Internet. Their goal is to achieve the universal interoperability between applications by using web standards, as well as to specify the technical infrastructure for carrying out business transactions. BPEL4WS (BPEL) is the OASIS standard for web services composition and orchestration. It contains several distinct features, including scope-based compensation and fault handling mechanism.

The important feature of BPEL is that it supports the long-running interactions involving two or more parties. Therefore, it provides the ability to define fault and compensation handing in application-specific manner, resulting in a feature called *Long-Running (Business) Transactions (LRTs)*. The concept of *compensation* is due to the use of Sagas [3,10] and open nested transactions [17].

* Corresponding author.

The fault analysis has been considered in [19] for compensation processes. Butler *et al.* have investigated the compensation feature in the style of process algebra CSP, namely *compensating* CSP [7]. The compensation is expressed as $P \div Q$, where P is the forward process and Q is its associated compensation behaviour. If a process encounters a fault some time after the execution of P , the scheduling of performing Q can undo the behaviour of P . In addition to the above two features, BPEL provides two kinds of synchronization techniques for parallel processes. In our model, shared-variables are introduced for the data exchange and the synchronization between a process and its partners within a single service, while channel communications are introduced for message transmission between different services.

Due to the interesting features of BPEL programs mentioned above, the verification of BPEL programs is challenging. Verification of shared-variable concurrent programs was first studied in [18]. The rely/guarantee approach is a compositional method for verifying concurrent programs [12]. The specification of the rely/guarantee method not only contains pre/post conditions, but also includes a rely-condition and a guarantee condition representing state changes made by the environment and the component respectively. Verification system for the rely/guarantee method has been studied in [20,22,9]. In this paper we apply the rely/guarantee method to verify BPEL programs.

The remainder of this paper is organized as follows. Section 2 introduces the syntax of BPEL programs and presents the operational semantics. In section 3, we provide the verification rules of BPEL programs using rely/guarantee method, including the rules for dealing with compensation, fault handling and shared-variable concurrency. Section 4 explores the soundness of the verification system. Section 5 discusses the related work about web services. Section 6 concludes the paper and presents some future work.

2 The Operational Semantic Model

2.1 The Syntax of BPEL

We have proposed a BPEL-like language, which contains several interesting features, such as scope-based compensation and a fault handling mechanism. Our language contains the following categories of syntactic elements:

$$\begin{aligned}
 BA &::= \text{skip} \mid x := e \mid \text{rec } a \ x \mid \text{rep } a \ x \mid \text{throw} \\
 A &::= BA \mid g \circ A \mid A; A \mid A \triangleleft b \triangleright A \mid \text{while } b \ \text{do } A \mid A \parallel A \\
 &\quad \mid \text{undo } n \mid \{A?A, A\}_n
 \end{aligned}$$

where:

- The category BA stands for the basic activity. $x := e$ assigns the value of e to shared-variable x . **skip** behaves the same as $x := x$. Here, variables for assignment can be regarded as the shared-variable understanding between different flows within one service.

Activity **throw** indicates that the program encounters a fault immediately. In order to implement the communications between different services, two

statements are introduced; i.e., $\mathbf{rec} \ a \ x$ and $\mathbf{rep} \ a \ x$. Command $\mathbf{rec} \ a \ x$ represents the receiving of a value through channel a , where $\mathbf{rep} \ a \ x$ represents the output of value x via channel a .

- The category A stands for the activities within one service. Several constructs are similar to those in traditional programming languages. $A; B$ stands for sequential composition. $A \langle b \rangle B$ is the conditional construct and $\mathbf{while} \ b \ \mathbf{do} \ A$ is the iteration construct. $g \circ A$ awaits the Boolean guard g to be set true, where event g is a Boolean condition.
- $\{A?C, F\}_n$ stands for the scope-based compensation statement, where A , C and F stand for the primary activity, compensation program and fault handler correspondingly. Here, n stands for the scope name. If A terminates successfully, program C is installed in the compensation list for later compensating. On the other hand, if A encounters a fault during its execution, the fault handler F will be activated. Further, the compensation part C does not contain scope activity. On the other hand, statement “undo n ” activates the execution of the programs with scope name n .

A service may contain one or several flows running in parallel. We use the notation $A \parallel B$ to stand for two processes running in parallel within one service. The parallel mechanism is the shared-variable model.

2.2 Operational Model

For the operational semantics of BPEL, its transitions are of the two types.

$$C \longrightarrow C' \quad \text{or} \quad C \xrightarrow{a.m} C'$$

where C and C' are the configurations describing the states of an execution mechanism before and after a step respectively. The first type is mainly used to model non-communication transitions. The second type is used to model the message communication between different services through channel a , where m stands for the message for communication.

The configuration can be expressed as $\langle P, \sigma, Cp \rangle$, where:

- (1) The first component P is a program that remains to be executed.
- (2) The second element σ is the state for all the variables.
- (3) The third element Cp stands for a compensation set; i.e., containing the scope names whose compensation parts need to be executed. Cp can contain several copies of the same element. Therefore, it can be understood as a bag. We use the scope name to identify the corresponding program that needs to be compensated. We introduce a function $C(n)$ to represent the program whose name (i.e., scope name) is n . When statement $\mathbf{undo} \ n$ is executed, process $C(n)$ will be scheduled and performed.

Regarding the program P in configuration $\langle P, \sigma, Cp \rangle$, it can either be a normal program. Further, it can also be one of the following special forms:

- ε : A program completes all its execution and terminates successfully. We use ε to represent the empty program.
- \boxtimes : A program may encounter a fault and stops at the fault state. \boxtimes is used to represent a program that is in the fault state.

2.3 Operational Semantics for BPEL

(1) Basic Commands

Firstly we list the operational semantics for basic commands. The execution of $x := e$ assigns the value of expression e to variable x , and leaves other variables unchanged.

$$\langle x := e, \sigma, Cp \rangle \longrightarrow \langle \varepsilon, \sigma[e/x], Cp \rangle$$

For communication commands, statement **rec a x** receives message m through channel a . The received message will be stored in variable x .

$$\langle \mathbf{rec\ a\ x}, \sigma, Cp \rangle \xrightarrow{a.m} \langle \varepsilon, \sigma[m/x], Cp \rangle$$

rep a x stands for the sending command of channel communication. In the following transition rule, $\sigma(x)$ stands for the data which needs to be set out.

$$\langle \mathbf{rep\ a\ x}, \sigma, Cp \rangle \xrightarrow{a.\sigma(x)} \langle \varepsilon, \sigma, Cp \rangle$$

throw encounters a fault immediately after activation, while leaving all variables and the compensation set unchanged.

$$\langle \mathbf{throw}, \sigma, Cp \rangle \longrightarrow \langle \boxtimes, \sigma, Cp \rangle$$

undo n does the compensation program corresponding to scope name n .

$$\langle \mathbf{undo\ n}, \sigma, Cp \rangle \longrightarrow \langle C(n), L, Cp \setminus n \rangle, \quad \text{where } n \in Cp$$

Here, function $C(n)$ represents the program whose name is n (i.e., the scope name). $Cp \setminus n$ represents that scope name n is removed once from Cp .

(2) Sequential Constructs

For sequential composition $P; Q$, if process P terminates successfully, the control flow will be passed to Q for further execution.

$$\text{if } \langle P, \sigma, Cp \rangle \xrightarrow{\beta} \langle P', \sigma', Cp' \rangle \quad \text{and } P' \neq \varepsilon, \boxtimes$$

$$\text{then } \langle P; Q, \sigma, Cp \rangle \xrightarrow{\beta} \langle P'; Q, \sigma', Cp' \rangle$$

$$\text{if } \langle P, \sigma, Cp \rangle \xrightarrow{\beta} \langle \varepsilon, \sigma', Cp' \rangle, \quad \text{then } \langle P; Q, \sigma, Cp \rangle \xrightarrow{\beta} \langle Q, \sigma', Cp' \rangle$$

On the other hand, if P encounters a fault during its execution, $P; Q$ also encounters a fault during its execution.

$$\text{if } \langle P, \sigma, Cp \rangle \xrightarrow{\beta} \langle \boxtimes, \sigma', Cp' \rangle, \quad \text{then } \langle P; Q, \sigma, Cp \rangle \xrightarrow{\beta} \langle \boxtimes, \sigma', Cp' \rangle$$

$g \circ P$ waits for the Boolean guard g to be set true through the update of variables.

$$\langle g \circ P, \sigma, L, Cp \rangle \longrightarrow \langle P, \sigma, L, Cp \rangle, \quad \text{if } g(\sigma)$$

For conditional and iteration, their transition rules are similar to the conventional programming language.

(3) Parallel Composition

Now we consider the transition rules for parallel composition. First we define function **par**(P, Q), which can be used in defining the transition rules for parallel composition. Let

$$\mathbf{par}(P, Q) =_{df} \begin{cases} \varepsilon & \text{if } P = \varepsilon \wedge Q = \varepsilon \\ \boxtimes & \text{if } P = \boxtimes \wedge Q = \boxtimes \vee P = \boxtimes \wedge Q = \varepsilon \vee P = \varepsilon \wedge Q = \boxtimes \\ P \parallel Q & \text{otherwise} \end{cases}$$

If one component performs a transition, the whole process can also perform the transition.

$$\text{If } \langle P, \sigma, Cp \rangle \xrightarrow{\beta} \langle P', \sigma', Cp' \rangle, \text{ then } \langle P \parallel Q, \sigma, Cp \rangle \xrightarrow{\beta} \langle \mathbf{par}(P', Q), \sigma', Cp' \rangle \\ \langle Q \parallel P, \sigma, Cp \rangle \xrightarrow{\beta} \langle \mathbf{par}(Q, P'), \sigma', Cp' \rangle$$

The function **par** indicates the program status for a parallel process after executing a transition. On the other hand, if both of the two components are in empty state, the whole process is also in empty state. If both of the two components are in fault state, or one is in fault state and another one is in empty state, then the whole parallel process is also in fault state.

(4) Scope

For scope $\{A?C, F\}_n$, if the primary activity A performs a successful transition which does not lead to the terminating state, the whole scope can also perform the successful transition of the same type.

$$\text{if } \langle A, \sigma, Cp \rangle \xrightarrow{\beta} \langle A', \sigma', Cp' \rangle \text{ and } A' \neq \boxtimes$$

$$\text{then } \langle \{A?C, F\}_n, \sigma, Cp \rangle \xrightarrow{\beta} \langle \{A'?C, F\}_n, \sigma', Cp' \rangle$$

Further, if the primary activity performs a transition leading to the fault state, the fault handler in the scope will be activated.

$$\text{if } \langle A, \sigma, Cp \rangle \xrightarrow{\beta} \langle \boxtimes, \sigma', Cp' \rangle, \text{ then } \langle \{A?C, F\}_n, \sigma, Cp \rangle \xrightarrow{\beta} \langle F, \sigma', Cp' \rangle$$

Furthermore, we give one auxiliary transition shown below. It indicates that the compensation program can be added into the compensation set when the primary activity has been terminated.

$$\langle \{\varepsilon?C, F\}_n, \sigma, Cp \rangle \longrightarrow \langle \varepsilon, \sigma, Cp \cup \{n\} \rangle$$

In order to support the proof of the soundness of our verification system (in Section 3) based on the operational semantics, we assume that every program is in an open environment. That is, every program can perform an environment transition (i.e., expressed as \xrightarrow{e}). For the above explored transition type \longrightarrow and $\xrightarrow{a.m}$, we generally regard them as \xrightarrow{c} (i.e., standing for the component transition) when studying the soundness of our verification system.

3 Verification Rules

In this section, we apply the rely/guarantee method to the verification of BPEL programs. The verification rules are in the form

$$P \text{ \underline{sat} } (pre, rely, guar, post)$$

Here, P stands for the program and pre stands for the pre-condition. After the execution of program P , post-condition $post$ should be satisfied. Furthermore, $rely$ and $guar$ are two relational formulae standing for the rely condition and

guarantee condition. The above notation indicates that

if P satisfies the precondition pre at the initial state
and any environment transition satisfies $rely$

then it should satisfy postcondition $post$ after the execution completes or encounters
a fault, and any component transition should satisfy $guar$.

We leave the proof of the soundness of our verification rules to the next section (i.e., section 4). For the aim of dealing with the two typical features of BPEL, i.e., fault handling and compensation, we introduce two variables ok and $comp$ in the four elements of a specification (pre , $rely$, $guar$, $post$).

- Boolean variable ok is used to identify whether a program is in the fault state or not. If ok is true (or false) in the pre-condition, this indicates that predecessor program is not in the fault state (or is in the fault state). On the other hand, if ok is true (or false) in the post-condition part, this indicates that the current system is not in the fault state (or in the fault state) after performing the program.
- In order to deal with the compensation feature, we need to record the number of each compensation program. Therefore, we introduce function $comp$ to handle this. More specifically, for the compensation program named n , we use $comp(n)$ to stand for the number that the compensation program has been recorded. Initially, for each scope n , its recoded number is 0; i.e., $comp(n) = 0$. For the compensated program named n , we use function $C(n)$ to represent it.
- For formula pre and $post$, they can contain variables ok , $comp$, as well as shared data variables x, y, \dots, z . On the other hand, $rely$ and $guar$ stand for the changes of data and compensation information due to the environment transition and component transition respectively. Hence, they should be in the form of relational formula, i.e., they should contain variables $comp, x, y, \dots, z$, as well as variables $comp', x', y', \dots, z'$. This indicates that $rely$ and $guar$ do not need to contain the information whether a process is in fault state or not, which means that they do not contain ok and ok' .

3.1 Starting from Fault State

Firstly, for every BPEL statement, we provide a general rule when a program is initially in the fault state. The rule indicates that if the current program starts in the fault state, it is always in the fault state, where the post-condition is the same as the corresponding pre-condition.

$$\frac{\begin{array}{l} pre \Rightarrow \neg ok \\ pre \wedge II \Rightarrow guar \\ pre \text{ \underline{stable when} } rely \end{array}}{S \text{ \underline{sat} } (pre, rely, guar, pre)}$$

where, (1) $II =_{df} (ok' = ok) \wedge (comp' = comp) \wedge (x' = x) \wedge \dots \wedge (z' = z)$

(2) $p \text{ \underline{stable when} } f =_{df} (p(V, ok) \wedge f(V, V')) \Rightarrow p(V', ok)$

Here, we use formula “ $pre \Rightarrow \neg ok$ ” to represent that a program is initially in the fault state. Furthermore, if a program is in the fault state, it cannot have chances to be executed, i.e., it cannot do anything. Our *guar* formula reflects this fact in the constraint condition “ $pre \wedge II \Rightarrow guar$ ”.

In the definition of *p stable when f*, variable vector *V* stands for the variable list, i.e., *comp, x, y, ..., z*, which indicates that formula *f* does not contain variable *ok* and *ok'*. It means that, after the environment transition reflected by *f*, *p* should also hold for the new updated program variables and compensation information due to the successful environment transition.

In the subsequent consideration, we explore the verification rules where the predecessor of the current program successfully terminates (i.e., not in the fault state).

3.2 Basic Commands

For **throw**, it immediately enters into the fault state while leaving the state of data variables unchanged. Therefore, for pre-condition $r \wedge ok$, the corresponding postcondition is $\neg ok \wedge r$.

$$\frac{\begin{array}{l} (ok \wedge III \wedge \neg ok') \Rightarrow guar \\ r \text{ does not contain variable } ok \\ (r \wedge ok) \text{ \underline{stable when} } rely \end{array}}{\mathbf{throw} \text{ \underline{sat} } (r \wedge ok, rely, guar, \neg ok \wedge r)}$$

Here, $III =_{df} (comp' = comp) \wedge (x' = x) \wedge \dots \wedge (z' = z)$. The execution of **throw** does not change the state of data variables. This can also be reflected from the constraint of *guar* in the form “ $(ok \wedge III \wedge \neg ok') \Rightarrow guar$ ”.

The rule for assignment is similar to traditional *rely/guarantee* methods when the program starts in the normal state.

$$\frac{\begin{array}{l} pre \Rightarrow ok \\ pre \Rightarrow post[e/x] \\ pre \text{ \underline{stable when} } rely \\ post \text{ \underline{stable when} } rely \\ pre \wedge ((x' = e \vee x' = x) \wedge \bigwedge_{s \in \{y, \dots, z, ok, comp\}} (s' = s)) \Rightarrow guar \end{array}}{x := e \text{ \underline{sat} } (pre, rely, guar, post)}$$

In this paper we focus on the *rely/guarantee* method within one service. Therefore, we do not study the verification rules for communication commands (**rec** *a x*, **rep** *a y* and their communication). Their investigation can be studied via the Hoare logic for process algebra [11].

3.3 Sequential Constructs

For sequential composition, there are two rules, which can be distinguished from the postcondition of the first program. The first rule below stands for the case

that the first program successfully terminates.

$$\begin{array}{c}
 pre \Rightarrow ok \\
 r \Rightarrow ok \\
 A \text{ \underline{sat}} (pre, rely, guar, r) \\
 B \text{ \underline{sat}} (r, rely, guar, post) \\
 \hline
 A ; B \text{ \underline{sat}} (pre, rely, guar, post)
 \end{array}$$

The second rule indicates that the first program for sequential composition encounters fault during its execution. Then the whole program will also be in the fault state. The postcondition for the whole program is the same as the postcondition for the first program.

$$\begin{array}{c}
 pre \Rightarrow ok \\
 r \Rightarrow \neg ok \\
 A \text{ \underline{sat}} (pre, rely, guar, r) \\
 \hline
 A ; B \text{ \underline{sat}} (pre, rely, guar, r)
 \end{array}$$

Now we consider the verification rules for $g \circ P$. The notation $g \circ P$ is used for the synchronization between different flows in a parallel composition.

$$\begin{array}{c}
 pre \Rightarrow ok \\
 pre \text{ \underline{stable when}} rely \\
 post \text{ \underline{stable when}} rely \\
 P \text{ \underline{sat}} (g \wedge pre, rely, guar, post) \\
 \hline
 g \circ P \text{ \underline{sat}} (pre, rely, guar, post)
 \end{array}$$

We can similarly consider the verification rules for conditional statement. Now we start to consider iteration. The rules for iteration can be divided into two cases. The first rule deals with the case when the iteration can perform the loop body several times and finally the loop condition will be unsatisfied.

$$\begin{array}{c}
 pre \Rightarrow ok \\
 pre \text{ \underline{stable when}} rely \\
 post \text{ \underline{stable when}} rely \\
 P \text{ \underline{sat}} (pre \wedge b, rely, guar, pre) \\
 \hline
 \text{while } b \text{ do } P \text{ \underline{sat}} (pre, rely, guar, pre \wedge \neg b)
 \end{array}$$

The second rule below can be used to indicate the case that the execution of iteration encounters a fault.

$$\begin{array}{c}
 pre \Rightarrow ok \\
 pre \text{ \underline{stable when}} rely \\
 post \text{ \underline{stable when}} rely \\
 post \Rightarrow \neg ok \\
 P \text{ \underline{sat}} (pre \wedge b, rely, guar, post) \\
 \hline
 \text{while } b \text{ do } P \text{ \underline{sat}} (pre, rely, guar, post)
 \end{array}$$

We also have the consequence rule.

$$\begin{array}{c}
 pre \Rightarrow pre_1, rely \Rightarrow rely_1, guar_1 \Rightarrow guar, post_1 \Rightarrow post \\
 P \text{ \underline{sat}} (pre_1, rely_1, guar_1, post_1) \\
 \hline
 P \text{ \underline{sat}} (pre, rely, guar, post)
 \end{array}$$

3.4 Scope and Compensation

Now we consider the verification rules for scope and compensation. These rules can represent the features of BPEL.

For scope activity $\{A?C, F\}_n$, the verification rules can be divided into two cases. The first rule explores the case that the primarily activity A can successfully terminate. Then the compensation program C needs to be recorded for later compensation. Therefore, for the specification of $\{A?C, F\}_n$, its precondition and rely condition are the same as those for primarily activity A respectively. The changes for guarantee condition and postcondition between A and $\{A?C, F\}_n$ are due to the recording of the compensation program.

$$\frac{\begin{array}{l} pre \Rightarrow ok \\ post \Rightarrow ok \\ A \underline{sat} (pre, rely, guar1, post[comp(n) + 1/comp(n)]) \\ guar1 \vee guar1[comp'(n) - 1/comp'(n)] \Rightarrow guar \end{array}}{\{A?C, F\}_n \underline{sat} (pre, rely, guar, post)}$$

The second rule below stands for the case that when A encounters fault, the fault handler F for the scope $\{A?C, F\}_n$ will be triggered. For the postcondition r of A , it should imply that ok is *false*. On the other hand, the fault handler F should be started at the normal state. Therefore, the precondition for F should be $r[\neg ok/ok]$.

$$\frac{\begin{array}{l} pre \Rightarrow ok \\ r \Rightarrow \neg ok \\ A \underline{sat} (pre, rely, guar, r) \\ F \underline{sat} (r[\neg ok/ok], rely, guar, post) \end{array}}{\{A?C, F\}_n \underline{sat} (pre, rely, guar, post)}$$

For *undo* n , the program corresponding to the name n (i.e., $C(n)$) will be executed. For the execution of compensation program $C(n)$, in the precondition of $C(n)$, the number of the recorded program named n should be one less, compared with the number of the recorded program named n before the execution of *undo* n .

$$\frac{\begin{array}{l} p \Rightarrow ok \\ comp(n) \geq 1 \\ guar1[comp'(n) - 1/comp'(n)] \vee guar1 \Rightarrow guar \\ C(n) \underline{sat} (pre[comp(n) + 1/comp(n)], rely, guar1, post) \end{array}}{\text{undo } n \underline{sat} (pre, rely, guar, post)}$$

3.5 Parallel Flows

In one service, flows are executed in parallel and communicate via shared-variables. The classic parallel rule can be applied, after modification of the postcondition to take into account the faulty states.

$$\begin{array}{c}
 pre \Rightarrow ok \\
 (rely \vee guar_1) \Rightarrow rely_2 \\
 (rely \vee guar_2) \Rightarrow rely_1 \\
 (guar_1 \vee guar_2) \Rightarrow guar \\
 P \underline{sat} (pre, rely_1, guar_1, post_1) \\
 Q \underline{sat} (pre, rely_2, guar_2, post_2) \\
 \hline
 P \parallel Q \underline{sat} (pre, rely, guar, Merge(post_1, post_2))
 \end{array}$$

$$Merge(q_1, q_2) =_{df} \exists ok_1, ok_2 \bullet q_1[ok_1/ok] \wedge q_2[ok_2/ok] \wedge ok = ok_1 \wedge ok_2$$

For $Merge(q_1, q_2)$, it not only combines q_1 and q_2 together with their program variables and compensation information, but also updates the state of ok for the whole system. The definition of $Merge(q_1, q_2)$ reflects the fact that the parallel system is in fault state if at least one component is in fault state.

4 Soundness of Verification Rules

In this section we start to study the soundness of verification rules. The soundness of our rely/guarantee approach is based on the operational semantics. Firstly, we give the definition of $A(pre, rely)$ and $C(guar, post)$, from which we can give the definition of the soundness for the rely/guarantee approach. Then we select some statements for studying the soundness of our verification rules, including sequential composition, scope and parallel composition.

4.1 Definition of Soundness

Definition 4.1 $C_0 \xrightarrow{\beta_1} C_1 \dots \dots \xrightarrow{\beta_n} C_n$ is a computation sequence, where $n \geq 1$. \square

Here, each C_i ($i = 0, \dots, n$) stands for a configuration. A computation sequence can represent the execution of a program.

We use the notation $cp[P]$ to denote the set containing all computation sequences of process P . Further, we use $cp[P]_{ter}$ and $cp[P]_{fau}$ to denote the set containing all computation sequences leading program P to the terminating state and fault state respectively.

For a computation sequence seq , we use notation $len(seq)$ to represent its length (i.e., the transition numbers). Notation $seq[i]$ is introduced to represent the i -th transition of seq , where $i \in \{1..len(seq)\}$. We also introduce notation seq_i to represent the i -th configuration of seq , where $i \in \{0..len(seq)\}$. Specially, notation seq_{last} is used to represent the last configuration of seq .

In order to give the soundness definition for our verification rules, we introduce the notations below.

Definition 4.2.

$$\begin{array}{l}
 A(pre, rely) \\
 =_{df} \{seq \mid seq_0 \models pre \text{ and for any environment transition in } seq
 \end{array}$$

$$\begin{aligned}
& seq_i \xrightarrow{e} seq_{i+1}, (seq_i, seq_{i+1}) \models rely \} \\
& C(guar, post) \\
=_{df} & \{seq \mid \text{for any transition } seq_i \xrightarrow{c} seq_{i+1} \text{ in } seq, (seq_i, seq_{i+1}) \models guar, \\
& \text{and if } Pr(seq_{last}) = \varepsilon \vee Pr(seq_{last}) = \boxtimes, \text{ then } seq_{last} \models post \}
\end{aligned}$$

□

For a configuration $\langle P, \sigma, Cp \rangle$, the notation “ $\langle P, \sigma, Cp \rangle \models F$ ” represents that $\langle P, \sigma, Cp \rangle$ satisfies formula F . This means that F is satisfied under data value state σ , Cp and P . More specifically, if P is not in the fault state, it means that “ $ok = true$ ” in F , otherwise it means “ $ok = false$ ” in F . For a scope name n , the numbers of n stored in Cp stand for the number $comp(n)$ in F . For a transition $C \xrightarrow{\beta} C'$, the notation “ $C \xrightarrow{\beta} C' \models F$ ” has similar understanding, where the unprimed variables in F are based on C and the primed variables are based on C' . Meanwhile, we use $Pr(C)$ to represent the program part of configuration C .

Based on the computation sequence, $A(pre, rely)$ and $C(guar, post)$, now we give the definition for the soundness of verification rules.

Definition 4.3. $P \underline{sat} (pre, rely, guar, post)$ is sound
 $=_{df} cp[P] \cap A(pre, rely) \subseteq C(guar, post)$ □

The understanding for program P satisfying a specification $(pre, rely, guar, post)$ starts from the concept of computation sequence. If a computation sequence of program P belongs to $A(pre, rely)$, it should also belong to $C(guar, post)$. This indicates that, if a computation sequence seq of program P satisfies the precondition pre and the rely condition $rely$ of each environment transition in seq , then the computation sequence seq should also satisfy the guarantee condition $guar$ for each component transition in seq . Moreover, if a computation sequence leads the program to the terminating state or fault state, it should also satisfy the postcondition $post$.

In the following part, we start to prove the soundness of the verification rules. We select some statements for consideration, including sequential composition, scope, and parallel composition. Others are similar.

4.2 Sequential Composition

For the two verification rules for sequential composition, here we focus on the second one. We select the typical computation sequence for $P; Q$ which leads the program to the fault state and it is due to the fact that process P encounters a fault. For other computation sequences of $P; Q$, the proof is similar. Let

$$\begin{aligned}
seq1 &: \langle P, \alpha \rangle \xrightarrow{\beta_1} \langle P_1, \alpha_1 \rangle \cdots \cdots \xrightarrow{\beta_n} \langle \boxtimes, \alpha_n \rangle, \\
seq &: \langle P; Q, \alpha \rangle \xrightarrow{\beta_1} \langle P_1; Q, \alpha_1 \rangle \cdots \cdots \xrightarrow{\beta_n} \langle \boxtimes, \alpha_n \rangle.
\end{aligned}$$

Here, the computation sequence $seq1$ leads program P to the fault state. Due to the fault encountering of program P , process $P; Q$ will also encounter fault. This can be reflected by the computation sequence seq for $P; Q$ and Q will not be scheduled in this case.

Now we need to prove that, for the above computation sequence seq , if seq belongs to $cp[P; Q] \cap A(pre, rely)$, it should also belong to $C(guar, post)$.

$$\begin{aligned}
 & seq \in cp[P; Q]_{faul} \wedge seq \in A(pre, rely) && \{\text{Relationship of } seq \text{ and } seq1\} \\
 \Rightarrow & seq1 \in cp[P]_{faul} \wedge seq1 \in A(pre, rely) && \{P \text{ sat } (pre, rely, guar, post)\} \\
 \Rightarrow & \text{Each } \xrightarrow{c} \text{ transition in } seq1 \text{ satisfies } guar \wedge && \{\text{Relationship of } seq \text{ and } seq1\} \\
 & seq1_{last} \models post \\
 \Rightarrow & \text{Each } \xrightarrow{c} \text{ transition in } seq \text{ satisfies } guar \wedge && \{\text{Def of } C(guar, post)\} \\
 & seq_{last} \models post \\
 \Rightarrow & seq \in C(guar, post) && \square
 \end{aligned}$$

4.3 Scope

For scope $\{A?C, F\}_n$, there are two verification rules. Firstly we consider the proof for the soundness of the first rule. We take the following type of computation sequence for the proof. Others are similar. Let

$$\begin{aligned}
 seq1 &: \langle A, \sigma, Cp \rangle \xrightarrow{\beta_1} \langle A_1, \sigma_1, Cp_1 \rangle \cdots \cdots \xrightarrow{\beta_m} \langle \varepsilon, \sigma_m, Cp_m \rangle, \\
 seq &: \langle \{A?C, F\}_n, \sigma, Cp \rangle \xrightarrow{\beta_1} \langle \{A_1?C, F\}, \sigma_1, Cp_1 \rangle \cdots \cdots \xrightarrow{\beta_m} \langle \{\varepsilon?C, F\}, \sigma_m, Cp_m \rangle \\
 & \quad \longrightarrow \langle \varepsilon, \sigma_m, Cp_m \cup \{n\} \rangle.
 \end{aligned}$$

Here, $seq1$ is the computation sequence leading program A to the terminating state, and seq is the corresponding computation sequence for $\{A?C, F\}_n$. seq records compensation name n in Cp in the last transition.

Now we consider the proof based on the soundness definition.

$$\begin{aligned}
 & seq \in cp[\{A?C, F\}_n]_{ter} \wedge seq \in A(pre, rely) && \{\text{Relationship of } seq \text{ and } seq1\} \\
 \Rightarrow & seq1 \in cp[A]_{ter} \wedge seq1 \in A(pre, rely) && \{A \text{ sat } (pre, rely, guar1, post1)\} \\
 & && \{post1 = post[comp(n) + 1 / comp(n)]\} \\
 & && \{seq[m + 1] = \{\varepsilon?C, F\}, \sigma_m, Cp_m\} \\
 & && \quad \longrightarrow \langle \varepsilon, \sigma_m, Cp_m \cup \{n\} \rangle \\
 \Rightarrow & seq1 \in C(guar1, post1) && \{\text{Def of } seq\} \\
 & seq[m + 1] \models guar1[comp'(n) - 1 / comp'(n)] \\
 & seq_{m+1} \models post \\
 \Rightarrow & seq \in C(guar, post) && \square
 \end{aligned}$$

Next we consider the soundness proof for the second verification rule for $\{A?C, F\}_n$. We focus on the computation sequence leading process A to the fault state. Let

$$\begin{aligned}
 seq1 &: \langle A, \alpha \rangle \xrightarrow{\beta_1} \langle A_1, \alpha_1 \rangle \cdots \cdots \xrightarrow{\beta_u} \langle \boxtimes, \alpha_u \rangle, \\
 seq2 &: \langle F, \alpha_u \rangle \xrightarrow{\gamma_1} \langle F_1, \alpha_{u+1} \rangle \cdots \cdots \xrightarrow{\gamma_m} \langle F_m, \alpha_{u+m} \rangle. \\
 seq &: \langle \{A?C, F\}_n, \alpha \rangle \xrightarrow{\beta_1} \langle \{A_1?C, F\}_n, \alpha_1 \rangle \cdots \cdots \xrightarrow{\beta_u} \langle F, \alpha_u \rangle \\
 & \quad \xrightarrow{\gamma_1} \langle F_1, \alpha_{u+1} \rangle \cdots \cdots \xrightarrow{\gamma_m} \langle F_m, \alpha_{u+m} \rangle.
 \end{aligned}$$

Here, $seq1$ is the computation sequence leading program A to the fault state, and $seq2$ is the computation sequence of F due to the fault encountering of A . Hence, here seq is the computation sequence of $\{A?C, F\}_n$, combining $seq1$ and $seq2$. Below is the detailed proof for seq .

$$\begin{array}{ll}
seq \in cp[\{A?C, F\}_n] \wedge seq \in A(pre, rely) & \{\text{Relationship of } seq, seq1 \text{ and } seq2\} \\
\Rightarrow seq1 \in cp[A]_{faul} \wedge seq1 \in A(pre, rely) \wedge & \{A \underline{sat} (pre, rely, guar, r)\} \\
seq2 \in cp[F] & \\
\Rightarrow seq1 \in C(guar, r) \wedge seq2 \in cp[F] & \{\text{Relationship of } seq, seq1 \text{ and } seq2\} \\
\Rightarrow seq1 \in C(guar, r) \wedge & \\
seq2 \in cp[F] \wedge seq2 \in A(r[-ok/ok], rely) & \{F \underline{sat} (r[-ok/ok], rely, guar, post)\} \\
\Rightarrow seq1 \in C(guar, r) \wedge seq2 \in C(guar, post) & \{\text{Relationship of } seq, seq1 \text{ and } seq2\} \\
\Rightarrow seq \in C(guar, post) & \square
\end{array}$$

4.4 Parallel Composition

Now we consider the soundness proof for the verification rule of $P \parallel Q$. First, we consider the merging behaviour for P , Q and $P \parallel Q$. Let

$$\begin{array}{l}
MERGE(seq1, seq2, seq) \\
=_{df} \left(\begin{array}{l}
len(seq1) = len(seq2) = len(seq) \wedge \\
\forall i \in \{0..len(seq)\} \bullet (Pr_i(seq1) \parallel Pr_i(seq2) = Pr_i(seq) \wedge \\
\quad St_i(seq1) = St_i(seq2) = St_i(seq)) \wedge \\
\forall j \in \{0..len(seq)\} \bullet (lab(seq1[j]) = c \wedge lab(seq2[j]) = e \wedge lab(seq[j]) = c \vee \\
\quad lab(seq1[j]) = e \wedge lab(seq2[j]) = c \wedge lab(seq[j]) = c \vee \\
\quad lab(seq1[j]) = e \wedge lab(seq2[j]) = e \wedge lab(seq[j]) = e)
\end{array} \right)
\end{array}$$

Here, $Pr_i(seq)$ and $St_i(seq)$ stand for the program part and the state part of the i -th configuration in seq . Function $lab(l)$ stands for the transition type for transition l , i.e., c transition (component transition) and e transition (environment transition) (see page [176](#)).

The analysis for the computation sequence of $P \parallel Q$ can be divided into the following three cases.

- (1) A computation sequence leads $P \parallel Q$ to the fault state.
- (2) A computation sequence leads $P \parallel Q$ to the terminating state.
- (3) A computation sequence leads $P \parallel Q$ to the non-fault, non-terminating state.

Here, we only focus on the proof for the first case. The considerations for other cases are similar. Let $seq \in cp[P \parallel Q]$. There are three typical cases:

- $\exists seq1 \in cp[P]_{ter}$ and $seq2 \in cp[Q]_{faul}$ and $MERGE(seq1, seq2, seq)$
- $\exists seq1 \in cp[P]_{faul}$ and $seq2 \in cp[Q]_{ter}$ and $MERGE(seq1, seq2, seq)$
- $\exists seq1 \in cp[P]_{faul}$ and $seq2 \in cp[Q]_{faul}$ and $MERGE(seq1, seq2, seq)$

Here we only consider the first subcase. Assume $seq \in A(pre, rely)$. We have the following facts.

- (a) Each c transition in $seq1$ and $seq2$ satisfies $guar1$ or $guar2$ respectively.
- (b) Each e transition in $seq1$ and $seq2$ satisfies $rely \vee guar2$ or $rely \vee guar1$ respectively.
- (c) Each c transition in seq satisfies $guar$.

Now we proceed with the detailed proof.

$$\begin{array}{ll}
 seq \in cp[P \parallel Q]_{fauc} \wedge seq \in A(pre, rely) & \{\text{Relationship of } seq, seq1 \text{ and } seq2\} \\
 \Rightarrow seq1 \in cp[P]_{ter} \wedge seq2 \in cp[Q]_{fauc} & \{\text{Fact (b) and } rely \vee guar \Rightarrow rely_1\} \\
 \Rightarrow seq1 \in cp[P]_{ter} \wedge seq2 \in cp[Q]_{fauc} \wedge & \{rely \vee guar_1 \Rightarrow rely_2\} \\
 \text{Each } e \text{ transition in } seq1 \text{ and } seq2 & \{\text{Def. } A(pre, rely)\} \\
 \text{satisfies } rely_1 \text{ and } rely_2 \text{ respectively } \wedge & \\
 seq1_0 \models pre \wedge seq2_0 \models pre & \\
 \Rightarrow seq1 \in cp[P] \wedge seq2 \in cp[Q]_{fauc} \wedge & \{P \text{ sat } (pre, rely_1, guar_1, post_1)\} \\
 seq1 \in A(pre, rely_1) \wedge seq2 \in A(pre, rely_2) & \{Q \text{ sat } (pre, rely_2, guar_2, post_2)\} \\
 \Rightarrow seq1 \in C(guar_1, post_1) \wedge seq2 \in C(guar_2, post_2) & \{\text{Def. } C(guar, post)\} \\
 \Rightarrow seq1_{last} \models post_1 \wedge seq2_{last} \models post_2 & \{\text{Fact (c) } \wedge \text{Merge}(post_1, post_2)\} \\
 \Rightarrow seq \in C(guar, Merge(post_1, post_2)) & \square
 \end{array}$$

5 Related Work

Compensation is one typical feature for long-running transactions. StAC (Structured Activity Compensation) [5] is another business process modeling language, where compensation acts as one of its main features. Its operational semantics has also been studied in [4]. Meanwhile, the combination of StAC and B method has been explored in [6], which provides the precise description of business transactions. Bruni *et al.* have studied the transaction calculi for Sagas [3]. The long-running transactions were discussed and a process calculi was proposed in the form of Java API, namely Java Transactional Web Services [2].

π -calculus has been applied in describing web services models. Laneve and Zavattaro [13] explored the application of π -calculus in the formalization of the semantics of the transactional construct of BPEL. They also studied a standard pattern of Web Services composition using π -calculus. For verifying the properties of long-running transactions, Lanotte *et al.* have explored their approach in a timed framework [14]. A model of Communicating Hierarchical Timed Automata was developed where time was also taken into account. Model checking techniques have been applied in the verification of properties of long-running transactions.

The above related approaches applied techniques in the specification and verification of BPEL, including process algebra and model checking techniques. Meanwhile, theorem proving is another approach for verifying BPEL programs. Luo *et al.* have studied the verification of BPEL programs using Hoare logic [16]. A big-step operational semantics has been studied and a set of proof rules were proposed. They were proven sound with respect to the formalized semantics. However, their approach does not cover the shared-variable feature for parallel composition. Compared with the above approach [16], shared-variables are introduced in this paper for the data exchange and synchronization between a process and its partners within a single service. Therefore, our paper here adopts the rely/guarantee approach to verifying concurrent BPEL programs.

6 Conclusion

Compensation and fault handling are the two main features of BPEL. This paper has explored the verification of BPEL programs, especially the verification

of concurrent BPEL programs. We have applied the rely/guarantee approach in designing the verification rules. We have provided an operational semantics. A set of verification rules has been explored, especially those rules for compensation, fault handling and concurrent programs. The verification rules have been proved to be sound based on the operational semantics.

In the near future, we plan to further study verification for BPEL programs, including automatic verification. It is challenging to implement our proof system in Theorem Prover. A case study would also be interesting to explore based on our verification rules presented in this paper. Meanwhile, it is also challenging to explore the link between the rely/guarantee method and the denotational semantics [11] for BPEL programs.

Acknowledgement. This work is supported in part by National Basic Research Program of China (No. 2011CB302904), National High Technology Research and Development Program of China (No. 2011AA010101 and No. 2012AA011205), National Natural Science Foundation of China (No. 61061130541 and No. 61021004), and Macau Science and Technology Development PEARL project (No. 041/2007/A3).

References

1. Apt, K.R., Francez, N., de Roever, W.P.: A proof system for communicating sequential processes. *ACM Trans. Program. Lang. Syst.* 2(3), 359–385 (1980)
2. Bruni, R., Ferrari, G.L., Melgratti, H.C., Montanari, U., Strollo, D., Tuosto, E.: From Theory to Practice in Transactional Composition of Web Services. In: Bravetti, M., Kloul, L., Zavattaro, G. (eds.) *EPEW/WS-EM 2005*. LNCS, vol. 3670, pp. 272–286. Springer, Heidelberg (2005)
3. Bruni, R., Melgratti, H.C., Montanari, U.: Theoretical foundations for compensations in flow composition languages. In: *Proc. POPL 2005*, January 12–14, pp. 209–220. ACM, Long Beach (2005)
4. Butler, M.J., Ferreira, C.: An Operational Semantics for StAC, a Language for Modelling Long-Running Business Transactions. In: De Nicola, R., Ferrari, G.-L., Meredith, G. (eds.) *COORDINATION 2004*. LNCS, vol. 2949, pp. 87–104. Springer, Heidelberg (2004)
5. Butler, M.J., Ferreira, C.: A Process Compensation Language. In: Grieskamp, W., Santen, T., Stoddart, B. (eds.) *IFM 2000*. LNCS, vol. 1945, pp. 61–76. Springer, Heidelberg (2000)
6. Butler, M.J., Ferreira, C., Ng, M.Y.: Precise modelling of compensating business transactions and its application to BPEL. *Journal of Universal Computer Science* 11(5), 712–743 (2005)
7. 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.) *CSP25*. LNCS, vol. 3525, pp. 133–150. Springer, Heidelberg (2005)
8. Curbera, F., Goland, Y., Klein, J., Leymann, F., Roller, D., Thatte, S., Weerawarana, S.: *Business Process Execution Language for Web Service* (2003)
9. de Roever, W.-P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Zwiers, M.P.J.: *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press (2001)

10. Garcia-Molina, H., Salem, K.: Sagas. In: Proc. ACM SIGMOD International Conference on Management of Data, San Francisco, California, USA, May 27-29, pp. 249–259. ACM (1987)
11. He, J., Zhu, H., Pu, G.: A model for BPEL-like languages. *Frontiers of Computer Science in China* 1(1), 9–19 (2007)
12. Jones, C.B.: Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.* 5(4), 596–619 (1983)
13. Laneve, C., Zavattaro, G.: Web-Pi at Work. In: De Nicola, R., Sangiorgi, D. (eds.) TGC 2005. LNCS, vol. 3705, pp. 182–194. Springer, Heidelberg (2005)
14. Lanotte, R., Maggiolo-Schettini, A., Milazzo, P., Troina, A.: Design and verification of long-running transactions in a timed framework. *Science of Computer Programming* 73(2-3), 76–94 (2008)
15. Leymann, F.: Web Services Flow Language (WSFL 1.0). IBM (2001)
16. Luo, C., Qin, S., Qiu, Z.: Verifying BPEL-like programs with hoare logic. In: Proc. TASE 2008, pp. 151–158. IEEE Computer Society, Nanjing (2008)
17. Moss, J.: Nested Transactions: An Approach to Reliable Distributed Computing. PhD thesis, Department of Electrical Engineering and Computer Science. MIT (April 1981)
18. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. *Acta Informatica* 6, 319–340 (1976)
19. Qiu, Z., Wang, S.-L., Pu, G., Zhao, X.: Semantics of BPEL4WS-Like Fault and Compensation Handling. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 350–365. Springer, Heidelberg (2005)
20. Stølen, K.: A Method for the Development of Totally Correct Shared-State Parallel Programs. In: Groote, J.F., Baeten, J.C.M. (eds.) CONCUR 1991. LNCS, vol. 527, pp. 510–525. Springer, Heidelberg (1991)
21. Thatte, S.: XLANG: Web Service for Business Process Design. Microsoft (2001)
22. Xu, Q., de Roeper, W.P., He, J.: The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing* 9(2), 149–174 (1997)

Completing the Automated Verification of a Small Hypervisor – Assembler Code Verification*

Wolfgang Paul, Sabine Schmaltz, and Andrey Shadrin

Saarland University, Germany

{wjp, sabine, shavez}@wjpsserver.cs.uni-saarland.de

Abstract. In [1] the almost complete formal verification of a small hypervisor with the automated C code verifier VCC [2] was reported: the correctness of the C portions of the hypervisor and of the guest simulation was established; the verification of the assembler portions of the code was left as future work. Suitable methodology for the verification of Macro Assembler programs in VCC was given without soundness proof in [3]. A joint semantics of C + Macro Assembler necessary for such a soundness proof was introduced in [4]. In this paper i) we observe that for two instructions (that manipulate stack pointers) of the hypervisor code the C + Macro Assembler semantics does not suffice; therefore we extend it to C + Macro Assembler + assembler, ii) we argue the soundness of the methodology from [3] with respect to this new semantics, iii) we apply the methodology from [3] to formally verify the Macro Assembler + assembler portions of the hypervisor from [1], completing the formal verification of the small hypervisor in the automated tool VCC.

1 Introduction

Kernels and Hypervisors: kernels and hypervisors for an instruction-set-architecture (ISA) M run on processors with ISA M and have basically two roles

- the simulation/virtualization of multiple *guests* or *user virtual machines* of ISA M
- the provision of services for the users via system calls (e.g. inter process communication)

The salient difference between kernels and hypervisors is that under kernels guests only run in user mode, whereas under hypervisors guests are also allowed to run in system mode. Thus, hypervisors must implement two levels of address translations (either supported by hardware features like nested page tables or in software using shadow page tables), whereas kernels must only realize one such level.

Kernel and hypervisor verification comes in 3 flavours:

- Verification of the C code alone. A famous example is seL4 [5]. Because kernels and hypervisors cannot be written exclusively in C such a proof is necessarily incomplete, but we will see shortly that closing this gap is not hard.

* Work partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft XT project under grant 01 IS 07 008. Authors in alphabetic order.

- Complete verification at the assembler level. Examples are the pioneering work on KIT [6] and the current effort in the FLINT project [7]. Complete code coverage can be reached in this way, but, due to the exclusive use of assembler language, productivity is an issue.
- Verification of both the C portion and the non C portion based on a joint semantics of C + inline assembler [8,9] or C + Assembler functions. As already shown in [10] C portions and non-C portions of a kernel can be verified separately and the correctness proofs can then be joined into a single proof in a sound way. The same could be done to cover the non C portions of seL4. In an interactive prover like Isabelle, which is used in [9] and [5], formal work can directly follow the paper and pencil mathematics. A small extra effort is needed if we want to perform such work in an automated C code verifier like VCC.

For work applying formal methods specifically to hypervisors consider the Nova micro-hypervisor [11], the recent MinVisor verification effort [12], or the partial verification of the Microsoft Hyper-V hypervisor [13].

The baby hypervisor [1] virtualizes a number of simplified VAMP [14,15] (called *baby VAMP*, see Fig. 2) guest processors (*partitions*) on a sequential baby VAMP *host processor*. The baby VAMP ISA is a simplified DLX-ISA (which is basically MIPS). The simplified VAMP architecture this work is based on does not offer any kind of virtualization support. Privileged instructions of guests (running in system mode) cannot be executed natively on the host. Instead, any potentially problematic instruction (e.g. write to the page-tables, change of page-table origin) causes an interrupt on the host machine and is subsequently virtualized by the baby hypervisor (see Fig. 3).

The baby hypervisor guarantees memory separation of guests by setting up an address translation from *guest physical addresses* to *host physical addresses* by defining a *host page table* [16] for each guest. A host page table is composed with the respective guest page table (if the guest itself is running in user mode) by the baby hypervisor to form a *shadow page table* that provides the direct translation from *guest virtual addresses* to *host physical addresses*. Then, running the host processor in user mode with the page table origin pointing to the shadow page table is sufficient to correctly virtualize the guest – as long as the guest does not perform changes to its own page table. To detect this case, the baby hypervisor marks those pages containing the guest page table as read only in the shadow page table. In case of a write access to the guest page table, a page-fault interrupt occurs, which allows the baby hypervisor to correctly virtualize the guest updating its page table.

As illustrated in Fig. 1, guest machines virtualized by the baby hypervisor are represented by memory regions (data structures of the baby hypervisor implementation) of the host machine. *Process-control-blocks* (PCBs) correspond to register contents of not-currently-running guests. At the beginning of the interrupt handler of the baby hypervisor, guest registers are saved to their corresponding PCB, the function `hv_dispatch()` is called to virtualize the instruction causing the interrupt, and, at the end of the handler, guest registers are restored to the host machine registers.

Specifying assembler portions of code in a C verifier: C is a universal language, hence it can simulate any other language. In [1], in order to verify the correctness of the baby

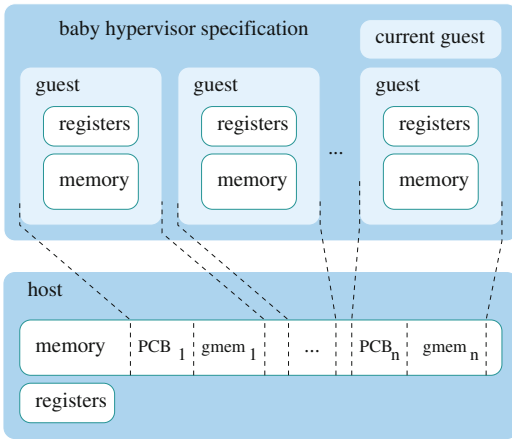


Fig. 1. Baby hypervisor overview

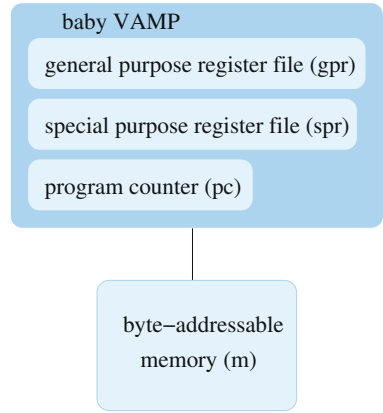


Fig. 2. Overview of the baby VAMP model

hypervisor, a baby VAMP interpreter is implemented in C. For the verification, the execution of hardware steps of the host processor that directly emulate guest steps are replaced by calls of the interpreter (which is implemented in such a way that it performs a single step of the baby VAMP model, i.e. executes a single machine instruction). The data structures of the interpreter are the obvious ones: i) hardware memory, which is fortunately already part of the C memory model of VCC and ii) processor registers, which are stored in a struct in a straightforward way. Based on this *and a specification of the effects of process save and restore*, the authors of [11] succeed to prove process separation of the guests. We are left with the problem to verify process save and restore in VCC and to integrate this proof with the existing formal proofs in a sound way. Consider that in the context of system verification, the notion of soundness encompasses that the resulting integrated formal model forms a sound abstraction of the physical machine's execution.

Verifying Macro Assembler in C: Theoretically, one could now try to prove properties of assembler programs in VCC by proving the properties for the interpreter running the resulting machine code programs. The expected bad news is that this turned out to be inefficient. The good news is that modern hypervisors tend to use Macro Assembler instead of assembler. The control operations of Macro Assembler (stack operations, conditional jumps to labels) fit C much better than the (unstructured) jump and branch instructions of assembler language. This in turn permits to perform a semantics-preserving translation of Macro Assembler code to C code. That this works in an extremely efficient way was shown in [3]. Indeed, in [17], the author reports about the (isolated) verification of all Macro Assembler portions of the Microsoft hypervisor Hyper-V. Thus, it seems that we are left with i) the task to formally verify the Macro Assembler portions of the small hypervisor from [11], ii) the task to integrate this into the formal proof reported in [11], and iii) to show that this is sound relative to a joint semantics of C + Macro Assembler presented in [4]. We achieve the first two tasks by extending the VCC proof of [11], and we provide a pencil-and-paper proof for the third task.

The Last Two Instructions: It turns out that two instructions of the hypervisor code are *not* compatible with the chosen Macro Assembler semantics: in our Macro Assembler, there is a built-in abstract notion of the stack. The corresponding stack pointer registers are not visible anymore on Macro Assembler level; they belong to the implementation. In order for the C + Macro Assembler code of the baby hypervisor to run properly when an interrupt is triggered on the host machine, however, they must be set up with appropriate values so that the stack abstraction for C + Macro Assembler is established. The good news is that this is done without using control instructions of the ISA, thus, the method from [3] can still be used. The moderately bad news is that in the end soundness has to be argued relative to a joint semantics of 3 languages: C + Macro Assembler + assembler.

Outline. In Section 2, we introduce a rather high-level stack-based assembler language that we call *Macro Assembler (MASM)*, not to be confused with Microsoft’s MASM) and merge it with a very low-level intermediate language for C, *C-IL*, yielding an integrated semantics of *C-IL* and *MASM* – which is amenable to verification with VCC. In order to justify that the integration of semantics is done correctly, we state compiler correctness simulation relations for the two languages as pencil-and-paper theory in Section 3.

Since the *MASM* semantics defined before in [4] is only applicable when the stack pointers have been set up correctly, we remedy this shortcoming by extending *MASM* semantics in a simple way suited specifically to the situation occurring in the baby hypervisor in Section 4. Having achieved full coverage of the baby hypervisor code with our stack-based semantics, we proceed by translating the *MASM* code portions to C code according to the Vx86-approach in Section 5 – using VCC to verify correctness of the translated code. We conclude with a brief discussion of the verification experience.

All details of theories presented in this paper can be found in [18].

2 Models of Computation

2.1 Macro Assembler – A Stack-Based Assembler Language

In [1], it is stated that the assembler code verification approach to be used for verification of the missing assembler portions should be the same that has already been used in the Verisoft project [8][15]: Correctness of assembler code execution is argued step-by-step on Instruction-Set-Architecture (ISA) level. Overall correctness in combination with the baby hypervisor C code is to be established by applying a compiler correctness specification that relates ISA and C configurations.

The assumption in [1] is that it would be quite simple to use the baby VAMP interpreter from [1] to verify the assembler code portions of the baby hypervisor by performing steps until the code has been executed. However, while this approach works decently in an interactive prover, this does not work so nicely in an automated prover. Running the baby VAMP interpreter for more than a few specific consecutive steps easily leads to huge verification times.

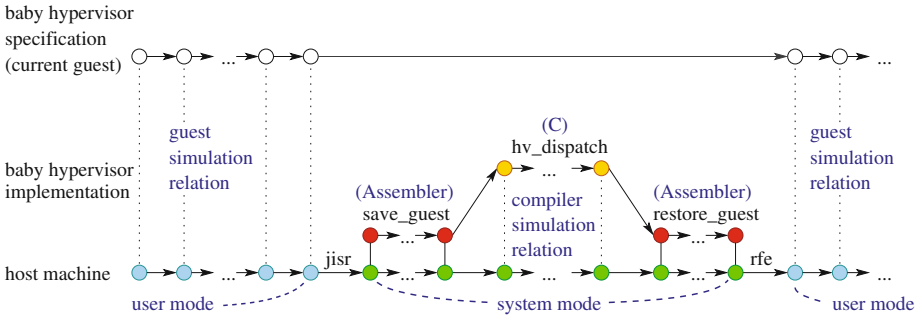


Fig. 3. Overview of baby hypervisor execution

With the tool Vx86 [3] it has been demonstrated that VCC can efficiently be used to verify (isolated) non-interruptible x86 Microsoft Macro Assembler code – by translating Macro Assembler to C which is verified with VCC. Non-interruptability can safely be assumed for the baby hypervisor code, thus, we decided to follow this approach. In order to formally argue soundness, we need an assembler code execution model for which simulation with a C code execution model is simple to establish. In this light comes our custom high-level stack-based assembler language we call *Macro Assembler* in the following brief summary.

Macro Assembler is a language with the following features: Jumps are expressed as (different flavors of) gotos to locations in functions, function calls and return are always made with the *call* and *ret* macros, and the memory region that holds the stack is abstracted to an abstract stack component (a list of stack frames) on which all stack accesses are performed. The first two choices restrict the applicability of MASM-semantics to well-structured assembler code. For baby VAMP, the following instructions are implemented as macros: *call*, *ret*, *push*, *pop*. A macro is simply a shorthand for a sequence of assembler instructions.

Configuration. A *Macro-Assembler* configuration

$$c = (c.\mathcal{M}, c.\text{regs}, c.s) \in \text{conf}_{\text{MASM}}$$

consists of a byte-addressable memory $\mathcal{M} : \mathbb{B}^{8k} \rightarrow \mathbb{B}^8$ (where k is the number of bytes in a machine word and $\mathbb{B} \equiv \{0, 1\}$), a component $\text{regs} : \mathcal{R} \rightarrow \mathbb{B}^{8k}$ that maps register names to their values, and an abstract stack $s : \text{frame}_{\text{MASM}}^*$. Each frame

$$s[i] = (p, \text{loc}, \text{saved}, \text{pars}, \text{lifo})$$

contains the name p of the assembler function we are executing in, the location loc of the next instruction to be executed in p 's function body, a component saved that is used to store values of callee-save registers specified by a so-called *uses list* of the function, a component pars that represents the parameter region of the stack frame, and a component lifo that represents the part of the stack where data can be *pushed* and *popped* to/from. For a detailed description of *Macro Assembler* semantics, see [18].

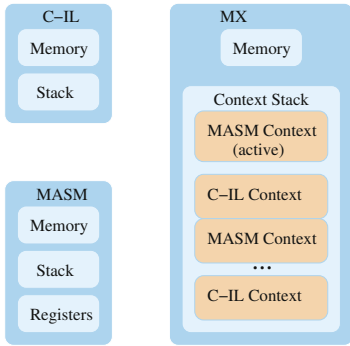


Fig. 4. Integrated semantics of C-IL and Macro Assembler with alternating execution contexts

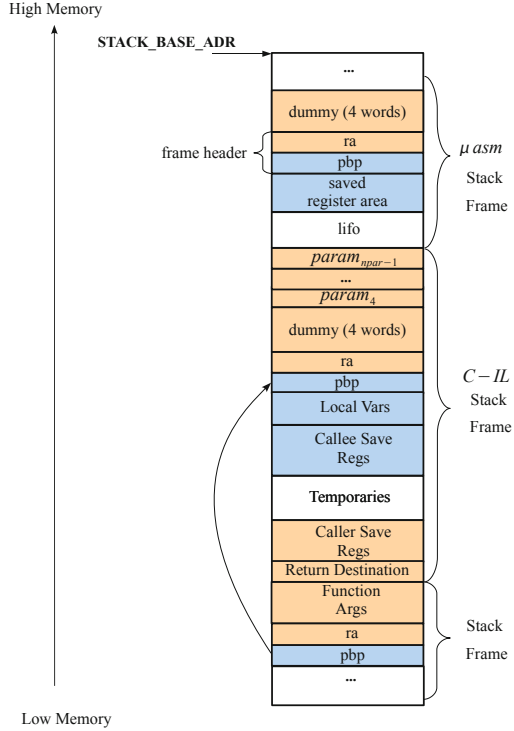


Fig. 5. MX-semantics stack layout

2.2 Integrated Semantics – Merging C Intermediate Language and Macro Assembler

In our verification effort, we are particularly interested in the correct interaction between assembler and C at function call boundaries. (Note that in contrast to the Verisoft project, we do not have inline assembler code here, but assembler functions calling C functions and vice versa.) Thus, we define an integrated semantics of a simple C intermediate language and *Macro Assembler* which allows function calls between those languages to occur according to the compiler’s calling conventions.

In order to describe this integrated semantics, we first give a very short overview of the features of our C intermediate language *C-IL*. *C-IL* is a very simple language that only provides the following program statements: assignment, goto, if-not-goto, function call, procedure call, and corresponding return statements. Goto statements specify destination labels. Pointer arithmetic on local variables and the global memory is allowed. There is no inherent notion of a heap in *C-IL*.

C-IL Configuration. A *C-IL* configuration

$$c = (\mathcal{M}, s) \in \text{conf}_{C-IL}$$

consists of a global, byte-addressable memory $\mathcal{M} : \mathbb{B}^{8k} \rightarrow \mathbb{B}^8$ and a stack $s \in \text{frame}_{C-IL}^*$ which is a list of *C-IL*-frames. Similar to *MASM*, a frame contains control information in form of location and function name – further, it contains a local memory that maps local variable and parameter names to their values as well as a return destination field for passing return parameters.

Integrated Semantics. In [4], we provide a more detailed report on the integrated semantics – a defining feature of which is its call stack of alternating *C-IL* and *MASM* execution contexts (see Fig. 4). Exploiting that both semantics use the same byte-addressable memory, we obtain a joint semantics in a straightforward way by explicitly modeling the compiler calling conventions and by calling the remaining parts of a *C-IL*- or *MASM*-configuration an execution context for the respective language.

Configuration. A mixed semantics (*MX*-) configuration

$$c = (\mathcal{M}, ac, sc) \in \text{conf}_{MX}$$

consists of a byte-addressable memory $\mathcal{M} : \mathbb{B}^{8k} \rightarrow \mathbb{B}$, an active execution context $ac \in \text{context}_{C-IL} \cup \text{context}_{MASM}$, and a list of inactive execution contexts $sc \in (\text{context}_{C-IL}^{\text{inactive}} \cup \text{context}_{MASM}^{\text{inactive}})^*$ which, in practice, is alternating between *C-IL* and *MASM*.

Here, an inactive execution context always contains information on the state of callee-save registers, which, before returning from *MASM* to *C-IL*, must be restored in order to guarantee that execution of compiled *C-IL* code will proceed correctly – or, respectively, the state of callee-save registers which will be restored automatically by the compiled *C-IL* code when returning from *C-IL* to *MASM*.

3 Compiler Correctness Specification

We assume a compiler correctness specification in the spirit of the C0 compiler [19] from the Verisoft project. We state a consistency relation that we expect to hold at certain points between a baby VAMP ISA- and a *MX*-computation (Figs. 6, 7).

Definition 1 (Code consistency). *The code region of the physical baby VAMP machine d is occupied by the compiled code of program p .*

$$\text{consis}_{\text{code}}(p, d) \equiv d.m_{\text{len}(\text{code}(p))}(\text{code}_{\text{base}}) = \text{code}(p)$$

where $\text{code}(p)$ denotes the compiled code of program p represented as a byte-string, len returns the length of such a string, $m_n(a)$ denotes reading a byte-string of length n starting at address a from byte-addressable memory m and $\text{code}_{\text{base}}$ denotes the address in memory where the code resides.

Definition 2 (Memory consistency). *The global memory content of the *MX*-machine c is equal to that of the physical baby VAMP machine d except for the stack and code region.*

$$\text{consis}_{\text{mem}}(c, d) \equiv \forall a \in \mathbb{B}^{32} \setminus \text{code}_{\text{region}} \setminus \text{stack}_{\text{region}} \quad : \quad c.\mathcal{M}(a) = d.m(a)$$

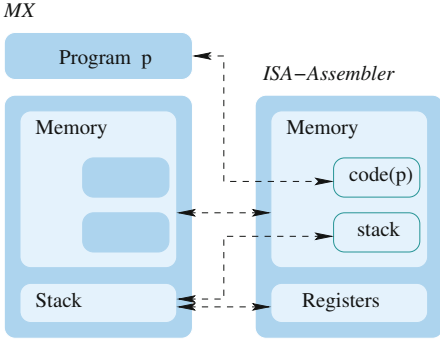


Fig. 6. Mapping abstract configuration to physical machine configuration by compiler consistency relation

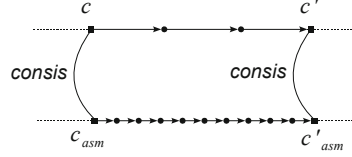


Fig. 7. Maintaining a compiler consistency relation between ISA- and MX-computation

Definition 3 (Stack consistency). *The stack component of the MX-machine c is represented correctly in registers and stack region of the baby VAMP machine d .*

$$\begin{aligned} \text{consis}_{\text{stack}}(c, d, p) \equiv & d.m_{\text{len}}(\text{flatten}_{\text{stack}}(c))(STACK_BASE_ADR) = \text{flatten}_{\text{stack}}(c) \\ & \wedge \text{consis}_{\text{regs}}(c, d, p) \end{aligned}$$

where $\text{flatten}_{\text{stack}}$ denotes a function that, given a MX-configuration returns a list of bytes that represent the stack in the physical machine according to the compiler definitions and $\text{consis}_{\text{regs}}$ specifies that all machine registers have the values expected for the given abstract stack configuration of c . These can only be defined when additional information about the compiler is given: E.g. in order to compute the return address field of a stack frame, we need to know the address in the compiled code where execution must continue after the function call returns. The calling conventions detail where parameters are passed (e.g. in registers and on the stack), while the C-IL-compiler defines the order of local variables on the stack (and whether they are cached in registers for faster access). For an exemplary stack layout of our integrated semantics, see Fig. 5.

Definition 4 (Compiler consistency). *An MX-configuration c and a baby VAMP configuration d are considered to be consistent with respect to a program p iff code consistency, memory consistency and stack consistency are fulfilled.*

$$\text{consis}(c, d, p) \equiv \text{consis}_{\text{code}}(p, d) \wedge \text{consis}_{\text{mem}}(c, d) \wedge \text{consis}_{\text{stack}}(c, d, p)$$

Definition 5 (Optimizing compiler specification). *The compiler relates MX computations (c^i) and baby VAMP ISA computations (d^i) via two step functions $s, t : \mathbb{N} \rightarrow \mathbb{N}$ with the meaning that, for all i , MX-configuration $c^{s(i)}$ and ISA-configuration $d^{t(i)}$ are consistent*

$$\forall i : \text{consis}(c^{s(i)}, d^{t(i)})$$

in such a way that the step function $s(i)$ at least describes those states from the computation (c^i) which are about to perform an externally visible action or where such

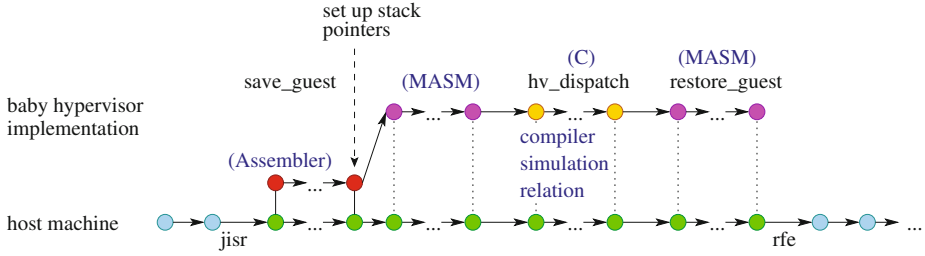


Fig. 8. The semantics stack applied to the baby hypervisor’s interrupt service routine after introduction of the first *Macro Assembler* version

an action has been performed in the previous step. Further, both s and t are strictly monotonically increasing.

In our current sequential setting without devices, the only externally visible action possible is an external function call. From the viewpoint of *C-IL*-semantics, calls to *MASM* functions are external, and vice versa.

For a well-structured example of a simulation proof for compiler correctness of a multi-pass optimizing compiler, see [20]. In the compiler specification sketched in the thesis of A. Shadrin [18], we expect compiler consistency to hold additionally at the beginning and at the end of function bodies, which – while restricting the extent of compiler optimization – simplifies the inductive proof significantly.

4 Extending the Semantics for Stack Pointer Setup

While we saw that the formalism of *Macro Assembler* is nicely suited to serve as a basis for justification of a translation-based assembler verification approach, it became obvious that the restrictions of *Macro Assembler* as described so far prevents the use of *Macro Assembler* for some parts of the baby hypervisor code. In fact, *Macro Assembler* semantics is a sound abstraction for execution of all but the first 46 assembler instructions of the baby hypervisor code – the last two of those 46 set up the stack pointer registers in order to establish the stack abstraction of *Macro Assembler* (see Fig. 8). Before those two instructions that set up the stack pointers occur, the stack pointers are uninitialized and we cannot establish compiler consistency for the preceding *Macro Assembler* execution.

The Root of the Problem. In order to run the *MX*-machine (which makes use of a rather high-level stack abstraction), we need to establish the stack abstraction correctly on the physical machine in the first place. In order to apply our compiler correctness specification, we need to establish initial compiler consistency, of which stack consistency is one part. The first part of the baby hypervisor’s interrupt handler implementation actually has to set up the stack abstraction for the baby hypervisor code to run by writing the stack pointer registers of the baby VAMP machine (see Fig. 8). The stack abstraction of the original *Macro Assembler* semantics, however, abstracts the stack pointers away.

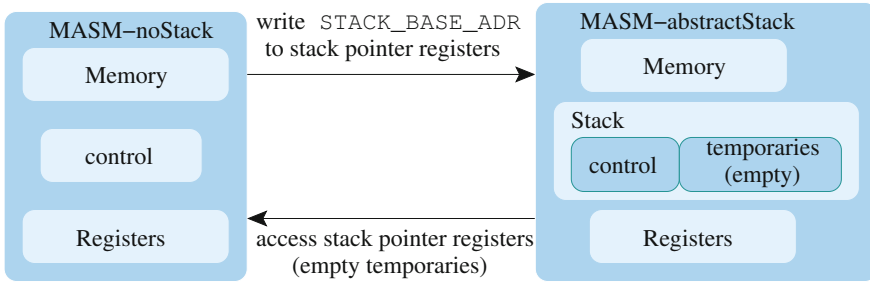


Fig. 9. Switching between mode without abstract stack and mode with abstract stack

While we could have proven the correctness of the assembler instructions up to the initialization of stack pointers using the baby VAMP ISA model in an interactive prover (which would have been bearable after rewriting the assembler code to perform stack initialization much earlier), we instead looked at the problem from the other side: What changes do we need to make to *Macro Assembler* semantics in order to "lift" these instructions to the *Macro Assembler* formalism so that we can verify their correctness with VCC? Considering the baby hypervisor implementation closely, we are in a situation where the stack pointers are always set up in the same way: by writing the stack base address `STACK_BASE_ADR` of the baby hypervisor to both stack pointer registers, effectively resulting in an empty initial stack configuration (i.e. there are neither parameters nor saved register values nor temporaries that can be *popped* from the stack). This is the case since a baby hypervisor execution context is always created by an interrupt, then the baby hypervisor performs emulation of a guest step and then the execution context perishes by giving up control to the guest.

Our Proposed Solution. For the situation in question, a simple band-aid is to just extend *Macro Assembler* semantics in such a way that there are two *execution modes*:

- *abstractStack*: The existing one with stack abstraction (stack pointer registers are hidden), and
- *noStack*: a mode without stack (stack pointer registers are accessible while function calls and stack operations are prohibited).

Defining the transitions between execution modes for this case is simple: When executing in *noStack*-mode, writing `STACK_BASE_ADR` to both stack pointer registers immediately results in an equivalent configuration in *abstractStack*-mode with empty stack content, whereas accessing the stack pointers in *abstractStack*-mode while the stack content is empty leads to an equivalent *noStack*-mode configuration (see Fig. 9).

With these definitions, we achieve full code coverage on the baby hypervisor with the resulting improved *MX*-semantics. While the chosen solution is rather specific, its simplicity raises the question whether there are more cases in which we can lift assembler instructions incompatible with stack abstraction to the *Macro Assembler*-level under certain conditions. We think that the stack switch operation present in thread switch implementations (substituting the stack pointers of the physical machine with the stack

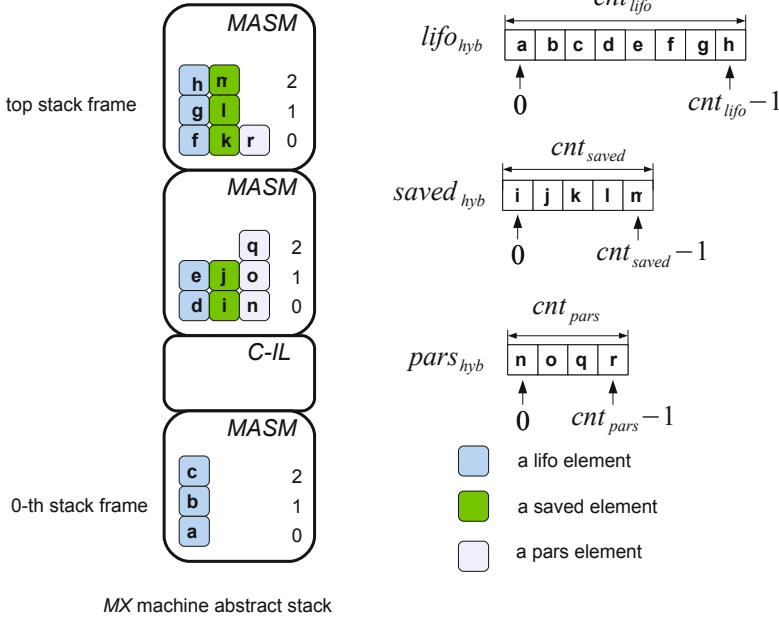


Fig. 10. Example of representing a *Macro Assembler* stack using C data structures

pointers associated with the next thread to run) is such a candidate, which will enable the sound verification of the code of thread switch implementations using automated C verifiers.

5 Assembler Verification Approach

For the assembler code verification of the baby hypervisor, we follow the general idea used in Vx86 [3]: Assembler code is translated to C code which is verified using VCC. Since there is very little assembler code in the baby hypervisor, we do not implement a tool that performs the translation (e.g., like Vx86) – instead, we formally define the translation rules and translate the code by hand according to the rules. We state this translation using *Macro Assembler* and *C-IL* semantics in [18].

The translation, in general, works as follows: we model the complete *MASM* state in *C-IL* using global variables and translate each *MASM*-instruction to one or several *C-IL*-statements. Recall that a *MASM*-configuration contains three main parts: the byte-addressable memory, a register component, and a stack of *MASM*-frames which each consist of control information, *saved* registers, parameters, and a component *lifo* of temporarily pushed to the stack. The byte-addressable memory is represented by *C-IL*'s own byte-addressable memory. For register content, we introduce global variables gpr and spr as arrays of 32-bit integer values of appropriate size. Control information is translated implicitly, by preserving the structure of function calls and jumps of the *MASM*-program during the translation. We model each of the stack components *saved*,

pars and *lifo* by a corresponding global 32-bit integer array variable and a 32-bit unsigned integer variable that counts the number of elements occupied in the array. For an example of representing *MASM*-state, see Fig. [10](#).

Translation. We define a function $\tau_{MX2IL} : Prog_{MX} \rightarrow Prog_{C-IL}$ which, given a program π_{MX} of the integrated semantics returns a *C-IL* program $\pi_{C-IL} = \tau_{MX2IL}(\pi_{MX})$. *MASM*-functions are translated to *C-IL*-functions; every *MASM* instruction of π_{MX} is translated to one or several *C-IL* statements while *C-IL* statements of π_{MX} are simply preserved in π_{C-IL} .

Consider the translation of the *push*-instruction:

$$\text{push } r \quad \Rightarrow \quad \text{lifo}_{\text{hyb}}[\text{cnt}_{\text{lifo}}] = \text{gpr}[r]; \text{ cnt}_{\text{lifo}} = \text{cnt}_{\text{lifo}} + 1$$

In *MASM* semantics, the *push*-instruction simply appends the value of register r to the *lifo*-component (which is modeled as a list in *MASM*-semantics and as an array with a counter in the translated program). Other examples are the translation of the *sw*-instruction or the *add*-instruction below:

$$\begin{aligned} \text{sw } rd \ rs_1 \ imm \quad &\Rightarrow \quad *((\text{int } *) (\text{gpr}[rs_1] + imm)) = \text{gpr}[rd] \\ \text{add } rd \ rs_1 \ rs_2 \quad &\Rightarrow \quad \text{gpr}[rd] = \text{gpr}[rs_1] + \text{gpr}[rs_2] \end{aligned}$$

Here, rd, rs_1, rs_2 are register indexes, and imm is a 16-bit immediate-constant – and *sw* is a *store word* instruction that stores the register content of register rd at offset imm of the memory address in register rs_1 , while *add* is an instruction that adds the values of registers rs_1 and rs_2 , storing the result in rd .

Soundness. A soundness proof for this approach is given by proving a simulation between the *MX*-execution of the original program π_{MX} and the *C-IL* program π_{C-IL} that results from the translation. The simulation relation and a pencil-and-paper proof are given in [\[18\]](#).

To achieve a clear separation between the original *C-IL*-code and the translated *MASM*-code, we place the data structures that model *MASM*-state at memory addresses which do not occur in the baby VAMP, i.e. addresses above 2^{32} . We call this memory region *hybrid memory*, since it is neither memory of the physical machine (which is covered by our compiler correctness specification) nor ghost state (VCC prevents information flow from ghost state to implementation state). Thus, the translation itself does not affect the execution of the original *C-IL* code parts. The actual proof is a case distinction on the step made in the *MX*-model: due to the way the translation is set up, correctness of pure *C-IL* steps is quite simple to show, while inter-language and pure *MASM*-steps have to explicitly preserve the simulation relation between the translated and the original program.

In order to formally transfer properties proven with VCC on the verified C code to the *MX*-execution we still lack a proof of property transfer from VCC C to *C-IL*. However, this gap should be straight-forward to close as soon as a formal model of VCC C is established.

The main advantage of how we implement this approach over how it has been done in the case of Vx86 is that we have a formal semantics for *Macro Assembler* that is quite similar to the C intermediate language (for which we also have formal semantics) we translate to. In [3], the closest formal basis for assembler code execution is given by the x86-64 instruction-set-architecture model developed by U. Degenbaev [21] – the authors apply the Microsoft Macro Assembler compiler to generate x86-64 code which is then translated to C. A monolithic simulation proof for this approach appears to be quite complex due to the big formal gap between the ISA model and the C semantics – thus, we deliberately chose *Macro Assembler* semantics as an abstraction of the ISA assembler code execution model in such a way that *Macro Assembler* semantics is structurally very similar to *C-IL* semantics.

6 Results and Future Work

Code Verification. The practical part of this work extends the code verification of the baby hypervisor by proving the central interrupt service routine correct – following its implementation in *Macro Assembler* (consisting of 99 instructions, most of them memory accesses to store/restore register values to/from the corresponding PCB). Translation of the *MASM* code results in approximately 200 additional C code tokens – the remainder of the baby hypervisor implementation consists of about 2500 tokens. For verification, an additional number of 500 annotation tokens were needed. Originally, about 7700 annotation tokens were present. With the formal models we have now, we believe that the actual code verification effort comes down to about one person week. It is quite obvious, that, from a practical verification engineering point of view, completing the baby hypervisor code verification was a minor effort compared to what already had been done. Our main contribution is the justification of this code verification.

Using the *verification block* feature of VCC, it was possible to keep verification times quite low (e.g. 41,48 seconds for the `restore_guest` function, 75,68 seconds for the `save_guest` function). This feature allows to split the verification of large C functions into blocks with individual pre- and postconditions. The total proof checking time of the completed baby hypervisor codebase is 4571 seconds (approx. $1\frac{1}{4}$ hours) on a single core of a 2.4 GHz Intel Core Duo machine.

Future Work. Possible extensions to this work include the generalization of calling conventions between *Macro Assembler* and *C-IL*. It appears desirable to have a semantic framework that can support many different compilers. For this, it could also be interesting to replace *C-IL* by a more mainstream intermediate language or a different flavor of C semantics. Similarly, *Macro Assembler* could be improved and generalized.

A work in progress deals with lifting the stack switch operation occurring in thread switch implementations to the *Macro Assembler* level. Extending the high-level semantics with a notion of active and inactive stacks (which is justified by a simulation with the ISA implementation layer), it should be possible to prove in an automated verifier that a given thread switch implementation based on switching stack pointers is correct.

7 Summary

In this work, we used an integrated semantics of *Macro Assembler*, a high-level assembler language, and *C-IL*, a simple C intermediate language, to give a pencil-and-paper justification of a translation-based assembler verification approach in the spirit of Vx86 [3]. In contrast to the original work, the translation is expressed rigorously based on formal semantics. We solved the problem of stack pointer setup by lifting a part of the ISA assembler semantics to our improved *Macro Assembler* semantics which we used as starting point for the translation of *Macro Assembler* code to *C-IL* code. The baby hypervisor implementation was completed by implementing the central interrupt service routine in *Macro Assembler* code, which was subsequently translated to C and verified with VCC – completing the formal verification of the baby hypervisor.

References

1. Alkassar, E., Hillebrand, M., Paul, W., Petrova, E.: Automated Verification of a Small Hypervisor. In: Leavens, G.T., O’Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 40–54. Springer, Heidelberg (2010), <http://www-wjp.cs.uni-saarland.de/publikationen/AHPP10.pdf>
2. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
3. Maus, S., Moskal, M., Schulte, W.: Vx86: x86 Assembler Simulated in C Powered by Automated Theorem Proving. In: Meseguer, J., Roşu, G. (eds.) AMAST 2008. LNCS, vol. 5140, pp. 284–298. Springer, Heidelberg (2008)
4. Schmaltz, S., Shadrin, A.: Integrated semantics of intermediate-language C and macro-assembler for pervasive formal verification of operating systems and hypervisors from VerisoftXT. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 18–33. Springer, Heidelberg (2012)
5. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP), pp. 207–220. ACM, Big Sky (2009)
6. Bevier, W.R.: Kit and the Short Stack. *J. Autom. Reasoning* 5(4), 519–530 (1989)
7. Feng, X., Shao, Z., Dong, Y., Guo, Y.: Certifying low-level programs with hardware interrupts and preemptive threads. In: 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008). ACM, New York (2008)
8. Verisoft Consortium: The Verisoft Project, <http://www.verisoft.de/>
9. Alkassar, E., Paul, W.J., Starostin, A., Tsyban, A.: Pervasive Verification of an OS Microkernel. In: Leavens, G.T., O’Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 71–85. Springer, Heidelberg (2010)
10. Gargano, M., Hillebrand, M., Leinenbach, D., Paul, W.: On the Correctness of Operating System Kernels. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 1–16. Springer, Heidelberg (2005), <http://www-wjp.cs.uni-sb.de/publikationen/GHLP05.pdf>
11. Tews, H., Weber, T., Völz, M., Poll, E., Eekelen, M., Rossum, P.: Nova micro-hypervisor verification formal, machine-checked verification of one module of the kernel source code (Robin deliverable d.13) (2008), <http://robin.tudos.org/>

12. Dahlin, M., Johnson, R., Krug, R.B., McCoyd, M., Young, W.D.: Toward the verification of a simple hypervisor. In: Hardin, D., Schmaltz, J. (eds.) *ACL2. EPTCS*, vol. 70 (2011)
13. Leinenbach, D., Santen, T.: Verifying the Microsoft Hyper-V Hypervisor with VCC. In: Cavalcanti, A., Dams, D.R. (eds.) *FM 2009. LNCS*, vol. 5850, pp. 806–809. Springer, Heidelberg (2009)
14. Tverdyshev, S.: Formal Verification of Gate-Level Computer Systems. PhD thesis, Saarland University, Computer Science Department (2009)
15. Tsyban, A.: Formal Verification of a Framework for Microkernel Programmes. PhD thesis, Saarland University, Computer Science Department (2009)
16. Alkassar, E., Cohen, E., Hillebrand, M., Kovalev, M., Paul, W.: Verifying shadow page table algorithms. In: *Formal Methods in Computer Aided Design, FMCAD 2010*, pp. 267–270. IEEE, Lugano (2010)
17. Maus, S.: Verification of Hypervisor Subroutines written in Assembler. PhD thesis, Freiburg University, Computer Science Department (2011)
18. Shadrin, A.: Mixed Low- and High Level Programming Language Semantics and Automated Verification of a Small Hypervisor. PhD thesis, Saarland University, Computer Science Department (to appear, 2012)
19. Leinenbach, D.: Compiler Verification in the Context of Pervasive System Verification. PhD thesis, Saarland University, Computer Science Department (2007)
20. Leroy, X.: Formal verification of a realistic compiler. *Communications of the ACM* 52(7), 107–115 (2009)
21. Degenbaev, U.: Formal Specification of the x86 Instruction Set Architecture. PhD thesis, Saarland University, Computer Science Department (2011)

A Configuration Approach for IMA Systems

Visar Januzaj¹, Stefan Kugele²,
Florian Biechele³, and Ralf Mauersberger⁴

¹ Technische Universität Darmstadt, Fachbereich Informatik
januzaj@cs.tu-darmstadt.de

² Technische Universität München, Institut für Informatik
kugele@in.tum.de

³ Cassidian Air Systems, New Avionics Structures
florian.biechele@cassidian.com

⁴ EADS Innovation Works
ralf.mauersberger@eads.net

Abstract. In this paper, we focus on system configurations of Integrated Modular Avionics (IMA) systems and present a novel approach for their calculation. We consider IMA systems based on ASAAC standards (STANAG 4626, EN 4660). These systems are modelled, by means of *blueprints*, using the SAE standardised modelling and analysis language AADL. For the calculation of system configurations, the required data is gathered from the system model and is transformed into a SAT modulo theory (SMT) formula. This formula includes a set of user input parameters, which steer the resource allocation. All feasible solutions satisfy the schedulability by a given set of scheduling schemes. The as schedulable considered configurations serve in choosing the final system configuration, for which a set of possible valid reconfigurations is calculated. To facilitate more compact allocations and increase the quality of (re-)configurations, we consider system *modes*. Both the chosen configuration and its corresponding reconfigurations are stored in the AADL system model, making all necessary data available within the same developing environment.

1 Introduction

Following the advances in modern technology, traditional avionics systems, so called *federated systems*, are faced with increased requirements and complexity, as well as increased development and maintenance costs. Furthermore, due to their long development process, federated systems are faced with the fact that upon completion they may end up running on outdated hardware/software components [1,2].

In order to cope with these issues, the concept of *Integrated Modular Avionics* (IMA) [3-8] originated. Being used at the beginning in a more conservative way, only on small subsystems, IMA has gained lately a wider usage in the avionics domain and by 2020 it is predicted that most of modern avionics systems will be applying IMA [9]. Contrary to federated systems, where each aircraft function (flight management, cruise speed control, autopilot, etc.) is coupled with

its special and uniquely designed hardware, which can be seen as *black boxes*, IMA systems are integrated into so called *cabinets of processors*, i. e., the hardware components of IMA systems are not dedicated to any particular function but serve as flexible hosts shared by multiple functions. Introducing new functions in a federated system would require additional black boxes, whereas for an IMA system it does not necessarily mean that new hardware is needed, thus contributing to component, weight, volume, and power consumption (CO₂ footprint) reduction. Furthermore, due to the independence between hardware and software any updates on either hardware or software do not affect the counterpart. In addition, the IMA concept makes the usage of COTS (components off-the-shelf) possible, allowing in this way a more flexible system development.

In general, IMA systems lead to significant reduction of development and maintenance cost as well as time to market of modern avionics systems [2-4,8,10]. However, safety being a crucial requirement, IMA systems introduce indeed new challenges, which were easier to cope with, played a minor role or were not present in federated systems. These are addressed/regulated by IMA guidelines (DO-297 [3]) and standards such as ARINC-653 [6] and ASAAC [4,5], that are used in the development of civil and military aircrafts, respectively. For instance, IMA systems must provide means, such as system partitioning [11], for fault containment for shared resources, being those processing or memory components, so that aircraft functions (software components) belonging to different criticality levels as categorised by the design assurance level in the DO-178B/ED12B [12] avionics standard—from no affect (level E) to catastrophic (level A)—do not affect each other, and the shared resources (hardware components) are sufficient enough for the correct execution of all aircraft functions, e. g., the software is provided with all necessary resources to meet its deadlines.

In this paper, we introduce a novel approach for the calculation of *system configurations* of IMA systems based on the *Allied Standard Avionics Architecture Council* (ASAAC) programme standards (NATO Standardization Agreement STANAG 4626 [4] and the European Standard EN 4660 [5]). The generated system configurations of such IMA systems, in following referred to as *IMA/ASAAC systems*, represent allocations/mappings of the software onto the underlying hardware. Our approach aims at facilitating the development of IMA/ASAAC systems and tackles the above mentioned IMA challenges by yielding schedulable system configurations, thus providing the allocated software with sufficient resources to meet their deadlines. The introduced methodology is applicable to ARINC-based IMA systems as well, as both standards indeed overlap and are similar. However, as pointed out in [13], the significant difference is that, while ARINC aims for *static* system partitioning (time and space), ASAAC is more flexible, thus facilitating *dynamic* system reconfigurations.

For the modelling of IMA/ASAAC systems, we employ the SAE (Society of Automotive Engineers) standardised modelling and analysis language AADL (Architecture Analysis and Design Language) [14,15]. For this purpose we developed a modelling concept that enables to capture the characteristics of an IMA/ASAAC system in corresponding AADL models, so called *blueprints*.

In order to calculate the system configurations, all the necessary data is extracted from the AADL models (blueprints) and is translated into an SMT formula. To improve the quality of the mapping between software and hardware components, a set of constraints is included into the SMT formula. As we are interested only on schedulable system configurations, all feasible solutions undergo a schedulability check, i. e., they are checked against schedulability requirements of scheduling algorithms such as EDF and RMS. The resulting feasible and schedulable configurations are then used to choose the final (optimal) system configuration. For the sake of *fault tolerance*, for the chosen system configuration a set of system *reconfigurations* is calculated, each introducing a scenario of a possible hardware failure. Both the finally chosen system configuration and its corresponding reconfigurations are written back in the AADL system model, thus providing all system data within the same developing environment. The reconfigurations are represented by means of a *state machine* introducing transitions from an unstable state, e. g., due to a hardware failure, into another stable state. This work provides the following contributions:

- (a) *Modelling*. In Sec. 3 we describe the modelling concept of IMA/ASAAC systems by means of AADL and show how such a system is organised as a blueprint.
- (b) *Flexibility*. Our configuration approach is introduced in Sec. 4. The configuration calculation (cf. Sec. 4.1) facilitates a flexible/tunable configuration mechanism for specifying different memory consumption and processor utilisation choices, e. g., requiring up to 70% overall resource utilisation.
- (c) *Schedulability*. By applying the schedulability check (cf. Sec. 4.1) only feasible configurations are generated. Furthermore, if desired by the user, an offline schedule plan for a chosen configuration is computed.
- (d) *Modes*. In Sec. 4.2 we extend our approach by considering system modes, and thus facilitating more resource-efficient configurations.
- (e) *Stability*. The reconfiguration calculation (cf. Sec. 4.4) tries to find reconfigurations that, when applied, lead to minimal changes with respect to the previous system state (prior reconfiguration).

We implemented our methodology as an Eclipse plug-in for OSATE¹ (Open Source AADL Tool Environment).

2 ASAAC

The main purpose of the ASAAC programme was to investigate the applicability of IMA on modern avionics systems and to implement corresponding standards in conformance with IMA (STANAG 4626, EN 4660), denoted *IMA/ASAAC standards*, satisfying the following goals [4, 5]: (i) *reduced life cycle costs*, (ii) *improved mission performance*, and (iii) *improved operational performance*.

In the following, we briefly highlight those concepts of IMA/ASAAC standards that are relevant for this paper. A more detailed description of these concepts

¹ <http://www.aadl.info>

and of IMA consideration in the ASAAC programme in general can be found in [4,5].

Software Architecture. To obtain the independence between the hardware and software components in an aircraft, being the most important characteristic of an IMA system, IMA/ASAAC standards follow a *three layer stack* concept for the software architecture of such systems: (i) *application layer*, (ii) *operating system layer*, and (iii) *module support layer*. The communication (and the independence) between the layers is established through standardised interfaces.

System Management. One of the major concepts of the IMA/ASAAC standards is the *system management*, which is responsible for the management and controlling of the aircraft system starting with its initialisation, during the whole flight and until the system is completely shut down. The system management consists of two parts and it spans both the application and the operating system layer. On the former layer resides the *application management*, and on the latter resides the *generic system management*, which has more responsibilities and provides means for specific important tasks, e. g., identifying and providing solutions in case of a fault occurrence: *health monitoring*, *fault management*, *configuration management* and *security management*. In order to achieve a better management performance, IMA/ASAAC systems include the following system management hierarchy: **Aircraft (AC) level** is the top level of an IMA/ASAAC system and is responsible for the management and controlling of the whole underlying system.

Integration Area (IA) level represents a subsystem consisting of the resources of closely related system applications and is responsible for the management and controlling of the subsystem it represents. An IA may be composed of further IAs.

Resource Element (RE) level is the lowest level of an IMA/ASAAC system and in the system hierarchy, and is responsible for the management and controlling of the hardware components.

Such a system (management) hierarchy is shown in Fig. 1. Each level of the hierarchy is managed by its corresponding system management component. However, the application management can be found only on the AC and IA levels, respectively, since RE is responsible for hardware only. Whereas the generic system management is present in all three levels. An IA can be seen as a partition, as the applications residing on an IA can be allocated only on the hardware managed by the REs belonging to this particular IA.

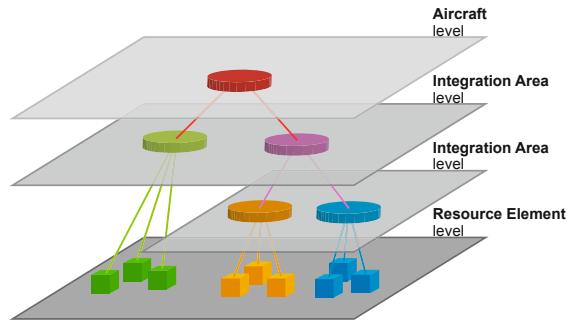


Fig. 1. The IMA/ASAAC system hierarchy

System Blueprints. The concept of *blueprints* is important for IMA/ASAAC systems as they define the whole system design and setup, and are used by the system management to manage and control the system. IMA/ASAAC standards distinguish between (i) *design time* and (ii) *run time* blueprints. The former are compiled at design time. The latter represent simply a translation of all the necessary system data, such as the mapping of software to the underlying hardware (allocation); the interconnection between software components and hardware components, respectively; and the partitioning [11] of the system, from the design time blueprints into a standardised database storage, forming the final system *blueprint*, which is loaded into the aircraft and is accessible at run time by the system management. A system blueprint represents a valid *system configuration* that according to the IMA/ASAAC standards “*is described in terms of configuration states and transitions between configuration states*” [4,5].

Reconfiguration. The flexibility of IMA/ASAAC systems supports *fault tolerance*, as required by the IMA/ASAAC standards, by means of the reconfiguration concept. A *reconfiguration* represents the transition of an IMA/ASAAC system at run time from one configuration to another, either as a result of an ordinary (initiated by the aircraft crew) or a fault-detection event. All possible reconfigurations should be calculated at design time and be included in the system blueprint (cf. guideline GUI-CR_15 in [4,5]).

Common Functional Modules. *Common functional modules* (CFMs) provide an IMA/ASAAC system with the necessary components to build the so called cabinets of processors that host the software components of an IMA/ASAAC system. Depending on their functionality, CFMs are categorised into six types (*signal processing module* (SPM), *data processing module* (DPM), etc.). Falling into the group of hardware components, CFMs belong to the RE level and are managed by their corresponding generic system management.

3 Modelling IMA/ASAAC Systems

To model IMA/ASAAC systems we apply the *Architecture Analysis and Design Language* (AADL) [14,15]. AADL is an SAE (Society of Automotive Engineers) standard that supports the specification of embedded systems through multiple abstraction levels, and it enables the application of various important system analyses such as behavioural and schedulability analysis. AADL components are divided into three categories:

Software represented by **subprogram**, **data**, **thread**, **thread group**, and **process**,

Hardware consisting of **device**, **memory**, **bus**, and **processor**, and

System where the composition of all component types is described in a (root) **system**, which can consist of further subsystems.

Each component is characterised via the *type* (or the interface) and its **implementation**. Further characteristics of the components and their relation to each other are given by means such as **features**, **properties**, **modes**, **port**,

connections, and **flows**. In addition, AADL components can *extend* other component's type/implementation, and can be organised in *packages*. Detailed information on AADL can be found in [14, 15].

In the following, we will represent the modelling of IMA/ASAAC systems by means of AADL. We will, however, focus only on the modelling of those parts of IMA/ASAAC systems that are relevant in our approach for the calculation of system configurations. Other parts, such as the detailed modelling of the system management, e. g., the generic system management and its functionalities (health monitoring, fault management, configuration management and security management), will not be discussed here.

First steps towards the use of AADL to model IMA/ASAAC systems were already made in 2004 [16], showing that the application of AADL facilitates a direct support of the IMA/ASAAC concept. In this paper, we push forwards the modelling of IMA/ASAAC systems in AADL by applying the concept of system *blueprints*. For this purpose we define four blueprint categories, each of them defined as a separate AADL **package**:

Application describes all software components (incl. subcomponents, e. g., the threads belonging to a process) and their requirements, such as required memory, communication specifics, deadline, period and execution time.

Resource describes all hardware components of the system platform, e. g., the type and available memory of CFMs, as well as the operating system (OS) that operates on.

Composition describes the composition between the components in the *Application* and *Resource* blueprints. Here the AC, IAs and REs and their sub-components are specified with respect to the IMA/ASAAC system management hierarchy.

System extends the *Composition* blueprint, respectively the AC system, with system allocations (given in terms of AADL predefined property **Actual_Processor_Binding**) and fault-triggered reconfigurations (given in terms of a state machine with **modes** and transitions between them).

Listing 1. AADL example

```

1 system Aircraft
2   properties
3     ASAAC_Properties::Elem_Type => AC;
4 end Aircraft;
5 system implementation Aircraft.impl
6   subcomponents
7     IA: system IA.impl;
8     RE: system RE.impl;
9     APP: system Application::Application.impl;
10    Platform: system Resource::Platform.impl;
11 end Aircraft.impl;
```

Components such as **AC**, **IA**, **RE**, **Application** (consisting of processes), **CFM** (consisting of processors and memory) and the **Platform** (consisting of CFMs and other AADL hardware components) are modelled each as an AADL **system**. The system states (excluding reconfigurations) are captured by means of an AADL state machine (**Statemachine**). Each state reflects a system mode (take off, navigation, landing, etc.).

We model the **StateMachine** and its modes as a **process** (residing on the *Composition* blueprint), which is extended by all other processes, thus inheriting the system modes. In order to be identified as such, the new components are annotated with a corresponding property given by **Elem_Type**, defined in the property set **ASAAC_Properties**, where further properties such as **CFM_Type** and **OS_Type** can be found.

A simplified version of the AADL model of an aircraft level (AC) is shown in Listing 1. In lines 1–4, where the *type* of the system **Aircraft** is specified, the ASAAC property **AC** is given, i. e., **Aircraft** is an AC, hence the top level of the whole system. Lines 5–11 show its *implementation*, where a number of subcomponents is specified. Each of these components is in turn a system of an ASAAC type (**IA**, **RE**, **Application** and **Platform**, respectively), and is designed similarly to **Aircraft**.

4 Configuration Approach

The idea behind our approach is depicted in Fig. 2. On the top we have the AADL blueprints of an IMA/ASAAC system, as well as the input parameters given by the user/system designer. The main process is shown in the lower part of the figure, namely the *(re-)configuration process*. In this part, *schedulable configurations* are calculated considering the blueprint constraints and the input parameters. Each of these configurations is checked against schedulability requirements and has successfully passed the *schedulability check* performed for the scheduling algorithms *earliest deadline first* (EDF) and *rate monotonic scheduling* (RMS), based on the processor utilisation conditions given by Liu and Layland [17]. Subsequently the configurations are ranked (cf. Sec. 4.3). The de-

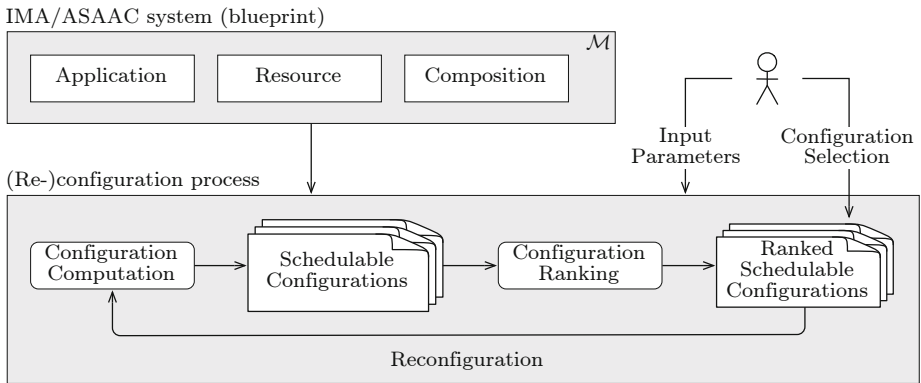


Fig. 2. Our (re-)configuration approach

cision of which configuration will be eventually chosen is solely made by the (experienced) system designer, who selects the preferred configuration and initiates the configuration computation again. However, the goal is now to calculate

possible *reconfigurations* for the originally selected one, by considering different topologies that reflect hardware malfunction. Both the chosen configuration and its valid reconfigurations are integrated into the AADL *System* blueprint (not shown in Fig. 2). In the following we will discuss in more detail each step of the configuration calculation process.

4.1 Calculation

The generation of possible system configurations is based on constraints ϕ extracted from system model \mathcal{M} , described by AADL blueprints.

Definition 1 (Allocation). *An allocation μ is a mapping of threads $t \in T$ to processors $p \in P$, i. e., $\mu : T \rightarrow P$, where T denotes the set of threads and P the set of processors in a system model \mathcal{M} , respectively. An allocation μ for model \mathcal{M} is called feasible, denoted $\mathcal{M}, \mu \models \phi$, iff it satisfies the constraints ϕ .*

Definition 2 (Configuration). *Let $\gamma = \langle \mathcal{M}, \mu \rangle$, $\gamma \in \Gamma$, be a system configuration, where \mathcal{M} is a system model, μ an allocation for \mathcal{M} , and Γ the set of all configurations. A configuration γ is called feasible if μ is feasible. $\Gamma_{\phi}^{\mathcal{M}}$ denotes the set of feasible configurations (the configuration suite) for system model \mathcal{M} .*

For a system model \mathcal{M} , we assume that the following constraints are encoded into the SMT formula ϕ (in this case the theory of linear integer arithmetic with Boolean constraints).

Constraint System. A feasible configuration γ has to ensure both (i) structural, denoted **Si**, and (ii) non-functional constraints, denoted **Ni**. Thus, in order to find feasible configurations, the generation of configurations has to consider both types of constraints, which are listed below:

- S1** (Partitioning) The allocation is performed with respect to the ASAAC system hierarchy (cf. Sec. 2), i. e., for instance, software components of an IA can only be bound/allocated to hardware components belonging to this IA (or its subsystem IAs).
- S2** (CFM_Type) Threads of a process that requires a certain CFM type, can only be allocated onto processors, whose parent module is of the respective CFM type.
- S3** (OS_Type) Threads of processes that belong to an application requiring a certain operating system (OS) type, can only be allocated onto processors, whose parent module provides the respective OS type.
- S4** (Locality) All threads belonging to the same process shall be allocated onto the same processor.
- N1** (Memory) The memory capabilities of modules shall not be exceeded by the memory requirements of processes, whose threads are bound to the processors of these modules.
- N2a** (Processor Utilisation) All threads bound to a processor shall not exceed the specified processor utilisation limit as given in Equation 2.

To generate the configuration suite, we use a technique similar to *iterative SAT solving* [18], but in contrast, we use SMT formulae, thus benefiting from both Boolean and mathematical reasoning. This approach generates iteratively a configuration suite $\Gamma_\phi^{\mathcal{M}}$, i.e., a set of configurations for system model \mathcal{M} satisfying the constraints ϕ . We start with an empty set of configurations $\Gamma_0 = \emptyset$ and iteratively add further configurations yielding $\Gamma_q \subseteq \Gamma_\phi^{\mathcal{M}}$ with $\Gamma_q = \{\gamma_1, \dots, \gamma_q\}$, $1 \leq q \leq m$ until we have reached a fixed point. This leads to $\Gamma_0 \subset \Gamma_1 \subset \dots \subset \Gamma_m = \Gamma_\phi^{\mathcal{M}}$ where m is the m -th iteration of the procedure.

Listing 2. Iterative Configuration Enumeration

```

1 func ICE( $\mathcal{M}, \phi$ )
2 begin
3    $q := 0$ ;
4    $\Gamma_q := \emptyset$ ;
5    $\text{CC}_0 := \phi$ ;
6   while  $\exists \gamma_{q+1} : \mathcal{M}, \mu_{q+1} \models \text{CC}_q$  do
7     begin
8        $\gamma_{q+1} := \text{getSolution}()$ ;
9        $\Gamma_{q+1} := \Gamma_q \cup \{\gamma_{q+1}\}$ ;
10       $\text{CC}_{q+1} := \text{CC}_q \wedge \neg \gamma_{q+1}$ ;
11       $q := q + 1$ ;
12    end;
13  return  $\Gamma_q$ ;
14 end;
```

We start in the first iteration with *configuration constraint* $\text{CC}_0 := \phi$, which only encodes the structural and non-functional requirements, leading to $\Gamma_1 = \{\gamma_1\}$ if there exists one. In the q -th iteration, the method provides—by construction—a new configuration γ_{q+1} if there is one. If in the q -th iteration no further configuration can be found, the fixed point has been reached. As long as there are new configurations, the procedure continues using γ_{q+1} to build the configuration constraint CC_{q+1} on basis of CC_q (following (1)).

$$\text{CC}_{q+1} := \text{CC}_q \wedge \neg \gamma_{q+1} \quad q \geq 0 \quad (1)$$

Adding the negated configuration result γ_{q+1} (cf. line 10), obtained using `getSolution()` in line 8, ensures that this solution is excluded from subsequent SMT solver solutions. In this way, all possible configurations are enumerated leading to the configuration suite $\Gamma_\phi^{\mathcal{M}}$ in the m -th iteration when the fixed point is reached, consisting of only unique configurations.

Schedulability Check. IMA/ASAAC systems profit from the configuration suite only if it contains *schedulable configurations*. Those configurations are feasible configurations and thus have to make sure that all deadlines are met. For this purpose, during the calculation process, we apply the processor utilisation $U(p)$ for processor p given by Liu and Layland [17]:

$$U(p) = \sum_{t \in T(p)} \frac{wcet_t}{\pi_t} \quad (2)$$

where $wcet_t$ is the execution time of thread $t \in T(p)$ on processor p , π_t is the period of t , and $T(p)$ is the set of threads residing on processor p .

Online/Dynamic Scheduling. In the case $U(p) \leq n(2^{1/n} - 1)$ ($\approx 0,7$), with $n = |T(p)|$, then the configuration is schedulable for RMS. In the case $U(p) \leq 1$ holds, we say the configuration is schedulable for EDF. This check is done for

each processor and the set of threads residing on them. The configurations not satisfying the processor utilisation condition are no further investigated and thus discarded.

Offline/Static Scheduling. If all threads are periodic and an offline schedule for a configuration (satisfying $U(p) \leq 1$) is sought by the system designer, a *schedule plan* wrt. a schedulable configuration is automatically calculated (cf. Fig. 3). Similar to [19], we use the proof that the schedule plan is valid as provided by the hyper-period H given by Leung and Merrill [20], i. e., if for a given scheduling plan within the interval $[0, H]$ all deadlines are met, with $H = \text{lcm}(\pi_t \mid t \in T(p))$ defined as the *least common multiple* (lcm) of all thread periods π_t , then this applies as a proof that the schedulability of the configuration is feasible.

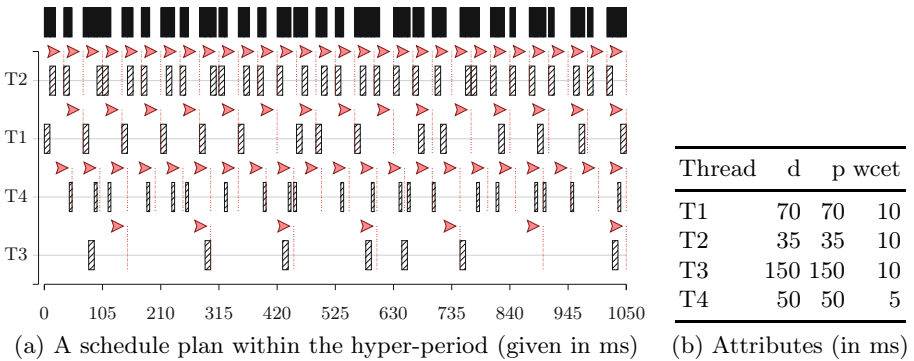


Fig. 3. Scheduling plan example

In Fig. 3a an example of such a scheduling plan is given, which shows four threads T1 to T4 allocated on a processor p with $U(p) \approx 0,59$ and hyper-period $H = 1050ms$. Their attributes are given in Fig. 3b. As both conditions (i) $U(p) \leq 1$ and (ii) all deadlines are met within the interval $[0, H]$, are satisfied, the given set of threads allocated on p is schedulable and the plan is valid.

4.2 System Mode Consideration

In order to achieve tight allocations, i. e., to use the available resources as good and efficient as possible, we extend our approach with the consideration of *system modes* (take off, navigation, landing, etc.). This extension allows the allocation of a larger number of processes/threads into processors, as not all of threads are active at the same time or active on the same system mode, respectively. For this purpose we need to separately consider the processor utilisation (cf. Equation 2) for a processor p on each system mode $m \in M$, denoted $U(p, m)$, where M is the set of all system modes. Thus, for the same set of threads bound on the same processor p it holds $U(p, m) \leq U(p)$ and $\sum_{m \in M} U(p, m) \geq U(p)$.

When considering system modes, constraint **N2a** is replaced by the following constraint:

N2b (Modes) For each single mode $m \in M$, all threads bound to processor p and active on mode m shall not exceed the specified utilisation $U(p, m)$.

Let us consider the example in Table **3b** and assume that the threads T1 to T3 are active on system mode *take off* (T), and T2 and T4 on mode *landing* (L). Both utilisations $U(p, T)$ ($\approx 0,49$) and $U(p, L)$ ($\approx 0,38$) are less than $U(p)$ ($\approx 0,59$), allowing further threads to be allocated on p .

4.3 Configuration Ranking

For the ranking of configurations (currently) we consider memory (consumption) and processor utilisation. The ranking is based on a scoring system where for each configuration the median value of (i) all processor utilisations and the corresponding (ii) memory consumption is calculated and the configurations are ranked, respectively; the position in the rankings is considered as the ranking score (position = score). In the case that the scores for two (or more) configurations are equal, we compare the average, maximum and minimum value, respectively. For each configuration the ranking scores in (i) and (ii) are summed up yielding the final ranking score. If two (or more) configurations have the same score, these configurations are considered as equal wrt. (i) and (ii). The configuration ranking should be understood only as assistance for the system designer in choosing the final configuration. Thus, which configuration is finally chosen is merely at the discretion of the system designer.

For example, consider configurations C_0 with scores: 5 in (i) and 1 in (ii) and C_1 with: 3 in (i) and 2 in (ii). Adding up the scores we get for $C_0 : 5 + 1 = 6$ and for $C_1 : 3 + 2 = 5$, thus C_1 is ranked higher, thus in a better position than C_0 .

4.4 System Reconfiguration

Besides the computation of possible system configurations, the presented approach is also best suited to generate *reconfigurations* (cf. Sec. **2**). Reconfigurations are performed (at run time) based on statically, i. e., at design time computed configuration alternatives. These alternative configurations take into account different malfunction scenarios: in principle all possible combinations of the unavailability of single or multiple processors/modules can be considered.

In this approach, we pursue the objective of *stability*. That is, for a configuration γ , a proper reconfiguration $\gamma^\#$ with respect to a hardware malfunction is computed with a maximum of similarity. The procedure tries to keep those parts of the configuration γ the same that correspond to unaffected hardware entities. We distinguish the following cases: (i) breakdowns of processors, and (ii) breakdowns of modules containing processors. In the first case, threads are not allowed to be bound to the broken processors, and in the second case, threads are not allowed to be bound onto *any* processors of a broken module. The second case also subsumes the circumstance that a module cannot be reached due to a broken network link.

5 Prototype Evaluation

Our method is implemented as an Eclipse plug-in prototype for OSATE. The plug-in allows the system designer to easily specify the input parameters: which scheduling algorithm to use, should the modes be considered, is a schedule plan desired, and the *utilisation factors*. The utilisation factors are used to tune memory and processor utilisation (constraints **N1**, **N2a/N2b**), respectively, e. g., for RMS with a utilisation factor of 0.5 we get a processor utilisation of $\approx 0.7 \times 0.5 = 0.35$. All the given parameters are integrated into the constraint system (cf. Sec. 4.1).

Table 1. Results

| Algo Modes | 1.0 | 0.9 | 0.8 | 0.7 | 0.6 | 0.5 |
|------------|-------------|-------------|-------------|-------------|-------------|-------------|
| EDF OFF | $\geq 10^3$ | $\geq 10^3$ | 76 | - | - | - |
| EDF ON | $\geq 10^3$ | $\geq 10^3$ | $\geq 10^3$ | $\geq 10^3$ | $\geq 10^3$ | $\geq 10^3$ |
| RMS OFF | - | - | - | - | - | - |
| RMS ON | $\geq 10^3$ | $\geq 10^3$ | $\geq 10^3$ | $\geq 10^3$ | 37 | - |

highlight here the advantage of considering system modes on the configuration calculation process. In Tab. 1 the configuration results of a rather small system (with 32 threads and 6 processors) are shown. The first column shows the scheduling algorithm used (EDF or RMS), the second column indicates if system modes are considered (ON) or not (OFF). The columns 3 to 8 show the utilisation factor used. While during the configuration calculation, with EDF and modes OFF, only 76 configurations could be calculated for a utilisation factor of 0.8 and none for less than 0.8, with modes ON it was possible to even find more than 1000 unique configurations for all utilisation factors up to 0.5. Due to the nature of the chosen system, no configurations could be found for RMS with no mode consideration (OFF). However, considering modes (ON), it was possible to calculate configurations for RMS for all utilisation factors up to 0.6.

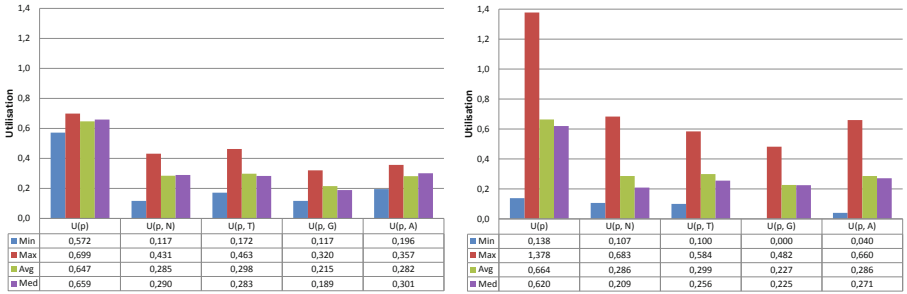
In Fig. 4 the best ranked (of 100) configurations (for RMS, with modes (ON)/(OFF) and factor 1.0) of a system with 116 threads and 9 processors are depicted, summarising the corresponding minimum, maximum, average and median values wrt. the processor utilisations $U(p)$ (first 4-column group) and $U(p, m)$ (the rest 4-column groups for each mode: *Navigation*(N), *TakeOff*(T), *GroundProcedure*(G) and *AutomaticLanding*(A)). Comparing the $U(p)$ values in Fig. 4a and Fig. 4b it can be noticed that the $U(p)$ in Fig. 4b exceeds the limit of 0.7, however, this shows only the $U(p)$ values corresponding to the actual allocations wrt. to each $U(p, m)$, as only the latter do not exceed the limit of 0.7. It can be also noticed that at least on one processor no thread is allocated in mode *GroundProcedure*. This allows more threads to be allocated on that specific processor.

The advantage of considering modes (cf. Tab. 1 and Fig. 4) is best observed when utilisation factors are used and most importantly during the reconfigu-

To evaluate the prototype, the IMA/ASAAC systems were automatically generated with different settings, as provided by our industrial collaborators, which are used as generation constraints for our evaluation purposes. We want to

ration calculation, where unallocated processes/threads, due to less available hardware, still have allocation possibilities as a result of $U(p, m) \leq U(p)$.

For the encoding of our constraint system, which has to be satisfied for feasible configurations, we proceed as follows: while constraints **S1** to **S3** can be checked statically by analysing the IMA/ASAAC system hierarchy, the remaining constraints are encoded into an SMT formula ϕ . The formula ϕ is then handed over to the SMT solver (in our case we used the award-winning high-performance SMT solver YICES [21]). Mostly for testing purposes, we have integrated into our prototype and explored two different encoding schemes: integer and division-weighted, and Boolean-weighted. The former scheme used 0/1-Integer and the latter Boolean indicator variables for the thread to processor mapping. To our surprise, the latter scheme turned out to lead to quite long calculation runs. Depending on the size and structure of the system, using the former encoding scheme the calculations were (in average) up to 60 times faster.



(a) No modes (OFF) and util. factor 1.0 (b) With modes (ON) and util. factor 1.0

Fig. 4. RMS scheduling

6 Related Work

There exists a wide variety of approaches that deal with modelling of real-time and safety-critical systems, as well as their scheduling. We will focus mainly on those that deal with AADL.

Delange et al. [19] presented an approach for modelling IMA systems in AADL with respect to the ARINC 653 [6] standard, using the ARINC 653 Annex of AADL. Furthermore, they introduce an efficient toolchain allowing amongst others to automatically perform schedulability analysis and corresponding C code generation. For scheduling they use the real-time scheduling framework Cheddar [22]. In [23], a similar approach for modelling IMA systems was introduced, where a different schedulability analysis methodology [24] was proposed.

Aleti et al. [25] developed the tool ArcheOpterix that applies evolutionary algorithms for optimising embedded systems architectures. These architectures can be modelled in AADL and for a given set of constraints, such as memory requirements and restrictions/permissions for processes to be allocated on the same processor, optimised system allocations are found.

De Niz et al. [26] presented a partitioning approach based on the bin-packing algorithm, which is efficiently applied for system allocations by keeping low the number of processors needed (with best-fitting allocations) as well as low off-board intercommunication between processes. The method is integrated in OS-ATE thus applicable to AADL system models [27].

Besides that none of the approaches mentioned above deal with IMA/ASAAC systems, none of them tackles the system configuration/reconfiguration and blueprint concept, nor do they provide means that facilitate flexible/tunable configuration calculation with respect to memory consumption and processor utilisation.

7 Conclusion

We have introduced in this paper a novel approach for the calculation of system (re-)configurations of IMA/ASAAC systems. For this purpose we designed a modelling concept for system blueprints of IMA/ASAAC systems using AADL. From the AADL blueprints we extract a set of constraints, which we encode into an SMT formula and the solutions of which represent the sought schedulable configurations. We include into the set of constraints input parameters given by the user/system designer, such as the specification of the scheduling algorithm to be considered and the utilisation factors, allowing the user to steer the configuration calculation. Furthermore, and most importantly, we consider system modes in the calculation process. As the evaluation of the prototype shows, this facilitates more compact allocations, and thus maximises the benefit from the potential of the underlying hardware, and increases the quality and the number of possible system (re-)configurations.

Currently we are preparing a real industrial case study to thoroughly evaluate our prototype. As future work it naturally fits into the agenda the extension of the framework in order to consider further constraints, e.g., communication load between processes, the extension of the modelling concept for IMA/ASAAC systems to include system management properties into AADL blueprints, as well as the consideration of software exclusion during reconfiguration calculation, e.g., in the case where only minimal resources (hardware and software) are available to perform a safe landing or to complete the mission.

References

1. Balser, B., Förster, M., Grabowski, G.: Integrated Modular Avionics with COTS directed to Open Systems and Obsolescence Management. In: RTO SCI Symposium on "Strategies to Mitigate Obsolescence in Defense Systems Using Commercial Components" (2001)
2. Brixel, T.: Transfer of advanced asaac sw technology onto the eurofighter/typhoon. In: 25th International Congress of the Aeronautical Sciences (September 2006)
3. RTCA DO-297: Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations (August 2005)
4. NATO Standardization Agreement: STANAG 4626: Parts I – VI
5. European Standard 4660: Aerospace series - Modular and Open Avionics Architectures, Parts 001–005 (2011)

6. ARINC 653-1: Avionics application software standard interface (October 2003)
7. Garside, R., Pighetti, F.J.: Integrating modular avionics: A new role emerges. In: IEEE/AIAA 26th. Digital Avionics Systems Conference, DASC 2007 (2007)
8. Cook, A.: ARINC 653 – Challenges of the present and future. *Microprocessors and Microsystems* 19(10), 575–579 (1995)
9. European Organisation for the Safety of Air Navigation (EUROCONTROL): Study report on avionics systems for 2011-2020 (February 2007)
10. Butz, H.: The Airbus approach to open Integrated Modular Avionics (IMA): technology, functions, industrial processes and future development road map. In: International Workshop on Aircraft System Technologies, Hamburg, Germany (2007)
11. Rushby, J.: Partitioning for avionics architectures: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center (June 1999)
12. RTCA DO-178B / EUROCAE ED12B: Software Considerations in Airborne Systems and Equipment Certification (1992)
13. Tiedemann, W.D.: Evaluation von Echtzeitbetriebssystemen für den Einsatz in Avioniksystemen (March 2008)
14. Society of Automotive Engineers: SAE Standards: Architecture Analysis & Design Language (AADL) - AS5506 (November 2004), and AS5506/1 (June 2004)
15. Feiler, P.H., Gluch, D.P., Hudak, J.J.: The architecture analysis & design language (aadl): An introduction. Technical report, SEI, Carnegie Mellon (2006)
16. Windisch, A.: Asaac modelling with aadl. SAE AS-2 Meeting on AADL (2004)
17. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM* 20(1), 46–61 (1973)
18. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
19. Delange, J., Pautet, L., Plantec, A., Kerboeuf, M., Singhoff, F., Kordon, F.: Validate, simulate, and implement arinc653 systems using the aadl. In: Proceedings of the ACM SIGAda Annual International Conference on Ada and Related Technologies, SIGAda 2009, pp. 31–44. ACM, New York (2009)
20. Leung, J.Y.T., Merrill, M.L.: A note on preemptive scheduling of periodic, real-time tasks. *Inf. Process. Lett.* 11(3), 115–118 (1980)
21. Dutertre, B., Moura, L.D.: The Yices SMT solver. Technical report, SRI International (2006)
22. Singhoff, F., Legrand, J., Nana, L., Marcé, L.: Cheddar: A flexible real time scheduling framework. In: Proceedings of the 2004 Annual ACM SIGAda International Conference on Ada, Atlanta, GA, USA (2004)
23. Januzaj, V., Mauersberger, R., Biechele, F.: Performance Modelling for Avionics Systems. In: Moreno-Díaz, R., Pichler, F., Quesada-Arencibia, A. (eds.) EUROCAST 2009. LNCS, vol. 5717, pp. 833–840. Springer, Heidelberg (2009)
24. Sokolsky, O., Lee, I., Clarke, D.: Schedulability analysis of aadl models. In: IPDPS. IEEE (2006)
25. Aleti, A., Bjornander, S., Grunske, L., Meedeniya, I.: Archeopterix: An extendable tool for architecture optimization of aadl models. In: ICSE Workshop on Model-Based Methodologies for Pervasive and Embedded Software, MOMPES 2009, pp. 61–71 (May 2009)
26. de Niz, D., Rajkumar, R.: Partitioning bin-packing algorithms for distributed real-time systems. *IJES* 2(3/4), 196–208 (2006)
27. de Niz, D., Feiler, P.H.: On resource allocation in architectural models. In: IEEE Int. Symp. on Object-Oriented Real-Time Distributed Computing, vol. 0, pp. 291–297 (2008)

polyLARVA: Runtime Verification with Configurable Resource-Aware Monitoring Boundaries

Christian Colombo¹, Adrian Francalanza¹, Ruth Mizzi¹, and Gordon J. Pace¹

Department of Computer Science, University of Malta
{christian.colombo, adrian.francalanza, rmiz0015, gordon.pace}@um.edu.mt

Abstract. Runtime verification techniques are increasingly being applied in industry as a lightweight formal approach to achieve added assurance of correctness at runtime. A key issue determining the adoption of these techniques is the overheads introduced by the runtime checks, affecting the performances of the monitored systems. Despite advancements in the development of optimisation techniques lowering these overheads, industrial settings such as online portals present new challenges, since they frequently involve the handling of high volume transaction throughputs and cannot afford substantial deterioration in the service they provide.

One approach to reduce overheads is the deployment of the verification computation on auxiliary computing resources, creating a boundary between the system and the verification code. This limits the use of system resources with resource intensive verification being carried out on the remote-side. However, under particular scenarios this approach may still not be ideal, as it may induce significant communication overheads. In this paper, we propose a framework which enables fine-tuning of the tradeoff between processing, memory and communication monitoring overheads, through the use of a user-configurable monitoring boundary. This approach has been implemented in the second generation of the LARVA runtime verification tool, polyLARVA.

1 Introduction

Due to its scalability and reliability, runtime verification [2] is becoming a prevalent technique for increasing the dependability of complex, security-critical, concurrent systems. Runtime verification broadly consists in checking for the correctness of the *current* system execution at *runtime*; it avoids checking alternative system execution paths and this, in turn, helps mitigate state-explosion problems associated with exhaustive techniques such as model checking. In scenarios where delayed correctness violation detection is unacceptable, runtime verification needs to be carried out in a *synchronous* fashion with the execution of the system. Invariably, synchrony increases the interaction with the system being monitored [1] and introduces overheads whose effects, particularly at peak times of system load, are hard to predict and may affect adversely the system behaviour.

¹ Although the monitoring for system events and the verification of the generated events against a specification are different operations, they are generally subsumed by the term *runtime verification* or *runtime monitoring* in the literature. We will use the terms interchangeably.

This issue is particularly relevant to scenarios such as online betting and e-commerce systems, where systems need to adequately handle multiple concurrent requests, resulting in uneven loads with peaks at particular times e.g., during an important sporting event. The sheer size and complexity of such online portals, compounded with the security-intensive nature of their execution (carrying a significant probability of fraud attempts) makes synchronous runtime verification a good candidate for increased runtime assurance. However, for this to be viable, the overheads introduced by the monitoring should not compromise the availability of system resources at any stage of the execution of the system.

A possible approach to reduce the system overheads introduced by runtime verification is to deploy the synchronous verification processes onto *separate computing resources*: events of interest generated by the system are sent over to the remote-side (resources), where the necessary monitoring computation takes place and, as soon as the *remote-side* deduces that the system has not violated the specification, control is returned back to the *system-side* thereby allowing the system to proceed. This approach promises to be effective when monitoring highly parallel systems because the synchrony required between monitor and system usually concerns only a *subset* of the system processes. Stated otherwise, in a system with a high degree of parallelism, the processes that are not covered by the current runtime check may continue executing unfettered on the system-side, without being burdened by the cost of the verification computation. In cases of resource-intensive monitoring, this cost in (monitored) system performance offsets any additional slowdown stemming from the added communication overhead introduced by the distributed monitoring architecture.

For instance, consider an e-commerce system handling transactions of users that are categorised as either greylisted (untrusted) or whitelisted (trusted) and a correctness property for each greylisted user involving a computationally expensive statistical analysis of all financial transfers performed by that user. Performing the greylisted user checks remotely frees the system-side from the associated computational overhead, ensuring that whitelisted users (and other greylisted users not performing a transfer at the moment) are not affected by the monitoring computation.

However, shifting all monitor checking to the remote-side is no silver bullet. In certain cases, performing verification checks remotely is impractical because these checks would require access to resources and state information kept by the system e.g., a local database; a remote evaluation of these verification conditions would require either resource replication or expensive remote access of resources. Furthermore, even when verification checks do not require access to system-side resources, there are still instances where shifting all runtime verification checks to the remote-side does not yield the lowest level of overhead. For instance, whenever the slowdown associated with communication outweighs the benefits gained from shifting monitoring checks to the remote-side, it is advantageous to perform the verification check at the system-side, circumventing any communication overhead. In the above e-commerce example, we may have a property stating that greylisted users may not transfer more than \$1000 in a single transaction: since the cost of a single integer comparison is typically less than that of communicating with the remote-side, it may pay to monitor the property on the system-side.

Even more complex situations may also arise. Consider again the e-commerce system which checks greylisted users for fraud. In this case, in order to minimise monitoring overhead, it may be advantageous to *split* the verification check across the system-side and remote-side. More precisely, in order to avoid communication overheads the check of whether a user is greylisted or not is performed on the system-side; this obviates the need for any communication with the remote-side (where the expensive statistical check is performed) whenever the user is whitelisted.

These examples attest that, in settings where verification can be done remotely, adequate control over where and when the verification computation is executed is essential for minimising the overheads of runtime monitoring security-critical concurrent systems. In spite of this need, the existing technologies that can be used for remote-verification (e.g., [8]) do not offer structuring mechanisms to support the fine-grained distribution control just discussed. In this paper, we propose a runtime verification system with a *configurable monitoring-boundary* enabling the user to decide which verification tasks are to be computed on the system side and which are executed on the remote side. This approach has been implemented in the tool polyLARVA, the second generation of the tool LARVA [5], where language-support is provided to enable the user to easily stipulate the system-monitor boundary. Such added flexibility empowers the user to decide the best allocation strategy for the runtime check at hand.

The contributions of the paper are:

1. the presentation of a framework enabling the fine tuning of system-side and remote-side computations;
2. showing the feasibility of the framework through its realisation in the polyLARVA tool; and
3. evaluating the approach through a number of case studies, measuring the effect of changing the configuration of the monitoring boundary.

The paper is organised as follows. In Section 2 we present our specification logic, the target architecture and the mapping from the logic to this architecture. Section 3 introduces the case study used to demonstrate the utility of our approach and Section 4 discusses the tests performed and the results obtained. Section 5 reviews related work and Section 6 concludes.

2 Configuring the Monitoring Boundary

We limit ourselves to an interpretation of runtime verification that consists of two main components. The behaviour of the system being monitored is characterised by a stream of *events* which are analysed by a *monitor* which may be composed of various monitoring tasks such as condition valuations and state updates. Our proposed framework provides mechanisms, in the guise of a monitoring boundary, for controlling the localisation of the monitoring tasks, when setting up the runtime verification configuration. One can therefore use this boundary to minimise the execution overhead introduced by the instrumented runtime verification on the system. An important caveat is that the partitioning of monitoring components between the system-side and remote-side largely depends on the kind of logic used and the forms of actions handled by the monitor.

For instance, if the logic is a simple one limited only to identifying undesirable events, *i.e.*, no temporal event ordering, then verification essentially requires little computation, which does not leave much scope for a monitoring boundary. However in more complex logics, such as LTL *e.g.*, [7][10], the monitor has to keep track of how much of the LTL formula one has already matched. Similarly, in logics expressed as symbolic automata *e.g.*, [5], one can directly program the monitor state; this gives more flexibility as to how much and which parts of the code are to be executed on either side. Analogously, the extent and the nature of the actions taken by the monitor also determines the level of placement manoeuvring that can be performed to minimise overheads.

2.1 Monitoring Using polyLARVA

Our monitoring framework polyLARVA uses a guarded-command style specification language. Properties are expressed as a list of rules of the following form:

$$event \mid condition \mapsto action$$

Whenever an *event* (possibly carrying parameters) is generated by the system, the list of monitor rules is scanned for rule matches relating to that event. If a match is found, the expression specified in the *condition* of the rule is evaluated and, if satisfied, the *action* is triggered.

Example 1. Consider a scenario in which one desires to monitor whether a greylisted user pays using a credit card after verifying it, blacklisting offending users if this rule is violated. This may be expressed in terms of the following two rules for each active user:²

$$\begin{aligned} register(user, card) \mid isGreylisted(user) &\mapsto registeredCards[card] := true; \\ pay(user, card) \mid isGreylisted(user) \wedge \neg registeredCards[card] &\mapsto blacklist(user); \end{aligned}$$

In the above rules, *register* and *pay* are system events, parametrised by the values *user* and *card*; *isGreylisted(user)* and $\neg registeredCards[card]$ are conditions, while *registeredCards[card] := true* and *blacklist(user)* are actions taken by the system. While some conditions and actions may be cheap to perform (*e.g.* *registeredCards[card] := true* consists of a single assignment), others may be more computationally expensive (*e.g.* a condition checking whether an ongoing transaction is fraudulent or not may require more computation).

2.2 The Target Architecture

Our approach gives sufficient high-level mechanisms such that a user can configure the monitoring boundary for synthesised lists of rules discussed in Section 2.1. This architecture allows for the possibility of having two sub-monitors that are automatically synthesised from a polyLARVA script, one on the system-side and the other on the remote-side as shown in Figure 1. Communication across nodes is carried out using socket connections: socket-based communication is low-level enough to be optimisable, while also providing the flexibility of technology-agnosticism.³

² For simplicity, we assume that the array of registered cards starts off with all cards unregistered, and that only a single user may access a particular card.

³ As future work polyLARVA will be extended to support multiple technologies.

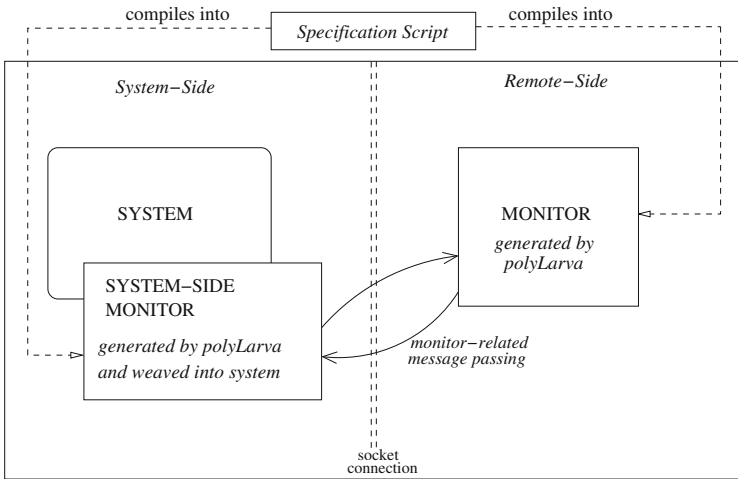


Fig. 1. Separation of monitor and system in polyLarva

The typical monitoring control flow over such an architecture proceeds as follows: (i) Events generated on the system-side trigger matching rules; (ii) Once a matching rule is found, the condition to be evaluated is determined. This condition is made up of a conjunctive sequence of (arbitrarily complex) basic conditions that are joined using normal negation, disjunction and conjunction operators. The basic conditions are categorised as either *system-side conditions*, i.e., predicates which are to be evaluated on the system-side, or *remote-side conditions*, i.e., predicates which should be evaluated at the remote-side; (iii) The evaluation of the condition starts and control is passed back and forth between the monitoring component on the system-side, whose role is that of evaluating the system conditions, and the monitoring component on the remote-side, whose role is to evaluate monitor conditions and coordinate control associated with the boolean operators. (iv) If the condition succeeds, the list of actions dictated by the matched rule is executed sequentially. Once again, actions are categorised as system- or remote-side, which entails passing control back and forth between the two sides; (v) Upon termination of the last action, control is passed back to the system to proceed.

Numerous variations and optimisations can be performed over the basic architecture of the above control flow. For instance, it may be beneficial to perform the rule matching process on the system-side, which avoids the overhead of communicating the event to the remote-side when a rule is not matched. In cases when the matched rule consists solely of system conditions and system actions, communication with the remote-side can be avoided altogether if the system-side monitoring component is entrusted with rule coordination capabilities. A similar situation may arise if part of the monitoring state is held at the system-side. Most of these optimisations are however case-dependent and do not apply to all verification specifications in general. Thus, such decisions cannot be automated: their control is best elevated at specification level and left to the verification engineer.

2.3 The Implementation

In polyLARVA the rules presented in Example 1 are coded as shown in Program 2.1. The rules are enclosed within a `rules` block, ①, which replicates them for every new user session that is opened, ②. The `state`, `conditions` and `actions` blocks define specification-specific monitor state, conditions and actions respectively that are used later in the rules section.

Program 2.1. Monitoring greylisted users for card registration

```

② upon (newUserSession(u)) {
  state {
    remoteSide { boolean[] registeredCards; }
  }
  conditions {
    systemSide { isGreylisted(u) = ... }
    remoteSide { isRegistered(c) = { registeredCards[c] } }
  }
  actions {
    systemSide { blacklistUser(u) = ... }
    remoteSide { registerCard(c) = { registeredCards[c] := true } }
  }
  ① rules {
    register(u,c) \ isGreylisted(u) -> registerCard(c);
    pay(u,c) \ isGreylisted(u) && !isRegistered(c) -> blacklistUser(u);
  }
}

```

In this example, `blacklistUser(u)` is a system action changing the internal state of the system relating to user `u`, whereas `registerCard(c)` is a remote action affecting the remote state `registeredCards` (a boolean array keeping track of which card numbers have been registered). Similarly, `isGreylisted(u)` is a system condition whereas `isRegistered(c)` is a remote condition, evaluated using the remote state `registeredCards`. As a result, the first rule's condition depends solely on system information, while the second refers to both the system state (whether a user is greylisted) and the remote state (whether the card was previously registered).

Deciding whether to place conditions and actions on the system-side or the remote-side to yield the minimum overhead is not straightforward and case-dependent. In the next section we use a real-life case study to investigate the alternatives.

3 Case Study

polyLARVA has been used to monitor JadaSite⁴ — an open source e-commerce solution written in Java, offering a range of features from back-office administration to automatic inventory control and online sales management. The case study focusses on

⁴<http://www.jadasite.com>

adding monitoring functionality to analyse user transactions in order to detect and prevent fraudulent transactions. Performing fraud detection in an offline manner, through the analysis of logs, is not ideal since by the time the analysis is carried out, a fraudulent user may have affected multiple transactions. Synchronous monitoring, enabling blocking a user upon suspicion of fraud, is desirable. Unfortunately, effective fraud detection typically requires costly analysis of the user’s history, involving multiple database accesses and processor-intensive calculations. Furthermore, since at peak times the system may have multiple users performing transactions concurrently, reducing the overhead is crucial.

The most straightforward partitioning is to place only parts which refer to the system state on the system-side, and placing the rest of the code on the remote-side. Through the code in Program 2.1 we see how polyLARVA allows the actions and conditions to be tagged in such a manner (highlighted in grey) instructing this code partitioning. However, partitioning of code can involve more intricate situations.

Example 2. Consider the following extension to Example 1 with the property that untrusted customers are not allowed to perform a payment if the probability of a fraud being committed is above a certain threshold.

$$\text{pay}(\text{user}, \text{card}) \mid \neg \text{isWhitelisted} \wedge \text{isFraudulent}(\text{user}) \mapsto \text{failTransaction}(\text{user}, \text{card});$$

The check for possible fraud is assumed to be a computationally expensive statistical analysis, while the decision of whether a customer is whitelisted is assumed to depend on the number of safely concluded payment transactions the user has already performed. The monitoring execution of this property is depicted in Figure 2[left], where the monitor is notified of the relevant events (*newUserSession* and *pay*), updates its customer state (increasing the number of transactions), checks any other appropriate conditions (checks the transaction with the customer history for fraud patterns), and performs actions accordingly (stopping the transaction) before returning control to the system. Program 3.1 shows an encoding of this example in polyLARVA, except for the location of state, conditions and actions, which will be discussed later.

Choosing the location of the monitoring code depends on different issues. For instance, since the fraud check ③ is assumed to be a resource-intensive operation, locating it on the remote-side relieves the system of the overhead, thus remaining responsive to the rest of the users. The stopping of a transaction ⑤ is an action which affects the system, meaning that it has to be located there. Finally, the transaction count is kept as a monitor state, ①, read by the monitor condition *isWhitelisted* ③ and written to by the verifier action *incrementTransactionCount* ⑥. This suggests that the three entities should be co-located so as to reduce additional communication for remote state access. Since the computation associated with this state is lightweight, it can be feasibly located on the system-side without affecting the system performance in any considerable manner. The script which specifies such a boundary configuration is shown in Program 3.2, with the resulting communication pattern shown in Figure 2[right].

Note that control may have to go back and forth the two sides multiple times due to the way the rules are structured. From this, it should be clear that (i) from a communication point of view, it would appear to be desirable to commute the two conjuncts on

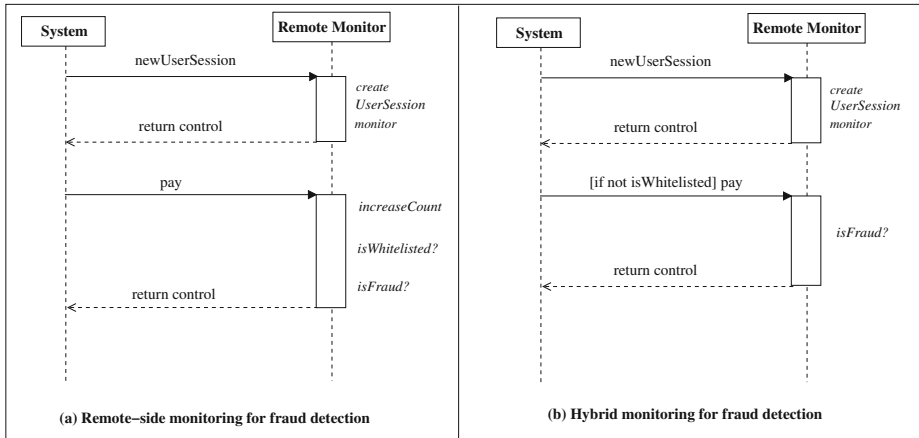


Fig. 2. Monitoring for fraud detection

Program 3.1. Monitoring for fraud detection

```

upon { newUserSession(u) } {
  state {
    ① integer transactionCount;
  }
  conditions {
    ② isWhitelisted = ...transactionCount...
    ③ isFraudulent(u) = ...CPU intensive algorithm...
  }
  actions {
    ④ incrementTransactionCount = transactionCount++;
    ⑤ stopTransaction(u,c) = ...
  }
  rules {
    ⑥ startTransaction \ true -> incrementTransactionCount;
    ⑦ pay(u,c) \ !isWhitelisted && isFraudulent(u) -> stopTransaction
      (u,c);
  }
}

```

Program 3.2. Tagging the case study with a monitoring boundary

```

② upon newUserSession(u) {
  conditions {
    systemSide {
      isWhitelisted(u) = {
        EntityManager em = JpaConnection...
        ... query = em.createQuery(sql);
        Long custTransNo = (Long) query.getSingleResult();
        return (...custTransNo.intValue()...)
      }
    }
    remoteSide { isFraudulent(u) = ...CPU intensive algorithm... }
  }
  actions {
    systemSide { failTransaction(u,c) = ... }
  }
  ① rules {
    pay(u,c) !isWhitelisted(u) && isFraudulent(u) -> failTransaction(u,c);
  }
}

```

line ⑦ so as to avoid control going from the monitor to the system side and back, but the high computational cost of checking for fraudulence means that we would prefer to start by performing the cheaper check for user trust first; (ii) If the system already keeps a record of payment transaction counts per user in its database, one may locate the trust checking condition to the system node, thus reducing communication by removing rule ⑥. In general, deciding the monitoring boundary can be seen as a minimisation problem having: a system-side and a remote-side, a number of (weighted) monitoring tasks, and a number of weighted communication signals as shown in Figure 3 (with task represented as boxes and communication as arrows). A minimal placement is one with the least number weighted boxes at the system-side, as few communication weights as possible, and having conditions which fail with a high probability as early as possible.

In the next section we give case study results corresponding to different monitoring boundary configurations showing how selected configurations can contribute to a non-negligible reduction of the monitoring overhead in a real-life case study.

4 Results

We have carried out a series of empirical tests showing how different monitoring configurations have a substantial impact on the performance of a monitored system. The noticeable overhead differences justify the need for a verification technique that permits flexibility with respect to the instrumented monitoring configurations *i.e.*, a configurable monitoring boundary. The tests also show how, in practice, the impact on system performance cannot always be fully predicted at instrumentation time. Thus, a level of abstraction that gives high-level control over the monitoring boundary, such as that

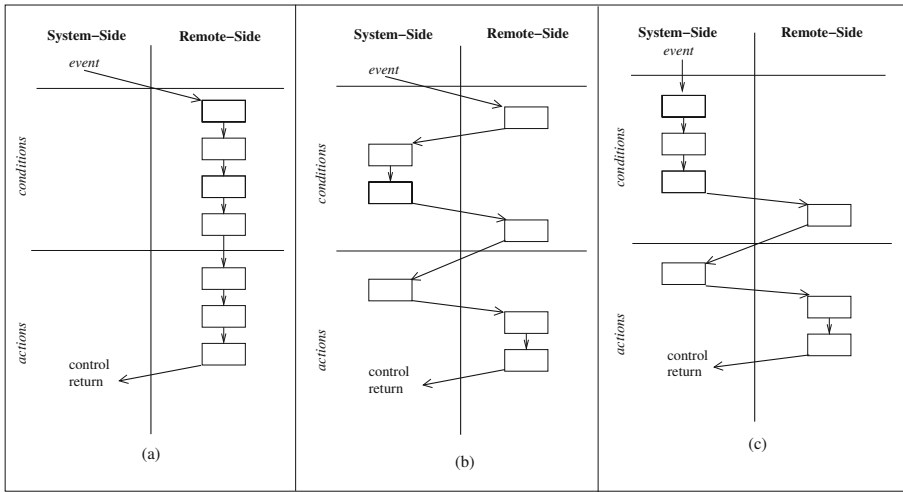


Fig. 3. Monitor placement

presented for polyLARVA, is required to facilitate the re-configuration of instrumented monitors.

Our tests are carried out on the JadaSite ecommerce system introduced in Section 3. Since the application we consider consists in a web portal handling an extensive amount of concurrent user request, an important aspect of system performance affected by monitor overheads is the Average time taken for a Payment Transaction (APT) to be processed, which directly translates to system responsiveness. Our tests synthesise runtime monitors for different monitoring boundary configurations for Program 3.1 using polyLARVA. Three distinct monitoring configurations were considered for our tests:

System-Side Monitoring (SM): Verification is entirely deployed on the system-side, running on the same address space as the system (all tags are *systemSide*).

Remote-Side Monitoring (RM): Verification checking exclusively is carried out on the remote-side, running on a separate machine from the system (all tags are *remoteSide*).

Hybrid Monitoring (HM): Verification checking is split between the system-side and the remote-side as in Program 3.2.

For each user-load level, we also benchmark the performance of the Unmonitored System (US), which helps us calculate the overhead introduced by each monitoring configuration.

The performance of the monitored system is benchmarked subject to user loads ranging from 40 to 120 concurrent user requests, involving operations such as adding items to the shopping cart, confirming payment details and executing the payment transaction. Load testing of the JadaSite web application, in conjunction with runtime monitoring, is carried out using Apache JMeter 2.5.⁵ Java SE 1.6 is used for the compilation of the

⁵ <http://jmeter.apache.org/>

JadaSite system source code and for the generation and compilation of the polyLARVA monitors. JadaSite is deployed on an Apache Tomcat 7.0.23⁶ server running on an AMD Athlon 64 X2 Dual Core Processor 6000+ PC, 4GB RAM, running Microsoft Windows 7. The remote-side consisted of a separate machine having an Intel(R) Core(TM)2 Duo CPU T6400, 2GB RAM with Microsoft Windows 7 operating system.

An important aspect affecting our tests is the ratio between whitelisted and greylisted users. This is because the verification checks specified in tests such as Programs 3.1 differentiate between whitelisted users and greylisted ones: greylisted users are subject to a monitoring condition requiring a computationally expensive fraud check whereas whitelisted users are not. For our experiments, two-thirds of the users are chosen to be whitelisted and the rest are considered to be greylisted. This ratio reflects more of a realistic deployment of the system where most of the users are regular users; the majority of these regular users are most likely to become whitelisted (trusted) after a probation period during which their transactions do not violate any verification checks.

Table 1. Average payment transaction duration for each user *wrt.* user load (in secs)

| Setup | 40 | 50 | 60 | 70 | 80 | 100 | 120 |
|-------|------------|------------|------------|------------|------------|------------|-------------|
| US | 11.4 | 17.7 | 24.0 | 28.7 | 37.4 | 56.2 | 69.2 |
| SM | 15.5 (35%) | 21.5 (21%) | 28.3 (17%) | 34.6 (20%) | 43.1 (15%) | 67.8 (21%) | 107.2 (55%) |
| RM | 13.5 (18%) | 19.7 (11%) | 26.8 (11%) | 34.4 (20%) | 41.9 (12%) | 60.4 (8%) | 89.9 (30%) |
| HM | 14.2 (24%) | 22.7 (28%) | 26.1 (8%) | 30.9 (8%) | 39.2 (5%) | 58.4 (4%) | 84.0 (21%) |

Table 2. CPU processing units used *wrt.* to user load

| Setup | 40 | 50 | 60 | 70 | 80 | 100 | 120 |
|-------|-------------|------------|-------------|-------------|--------------|--------------|----------------|
| US | 16598 | 21836 | 28340 | 34026 | 39416 | 58097 | 73266 |
| SM | 17591 (6%) | 22991 (5%) | 31315 (10%) | 36886 (8%) | 43295 (10%) | 63275 (9%) | 103850 (41.7%) |
| RM | 15215 (-8%) | 20971 (4%) | 26470 (-6%) | 33729 (-1%) | 42353 (7%) | 58981 (1%) | 93792 (28%) |
| HM | 16187 (-2%) | 23381 (7%) | 28333 (0%) | 32199 (-5%) | 41195 (4.5%) | 60636 (4.4%) | 86216 (17.7%) |

The main results of the experiments measuring the APT and the respective CPU usage at the system-side under different configurations can be found in Table 1 (depicted in Figure 4) and Table 2. When compared to the base APT of the unmonitored system in Table 1, it becomes evident that system-side monitoring (SM row) introduces substantial overheads, peaking at a level of 55% increase in APT when the tests hits a user load of 120 concurrent transactions. Table 2 indicates that the sharp increase in APT can be attributed to the increase in CPU usage at the system-side, depleting resources from the execution of the system.⁷ Such a deterioration in system responsiveness will most likely discourage the adoption of runtime verification checks over the live system.

⁶ <http://tomcat.apache.org/>

⁷ Fraud checking was not memory intensive and, as a result, memory usage was not an issue.

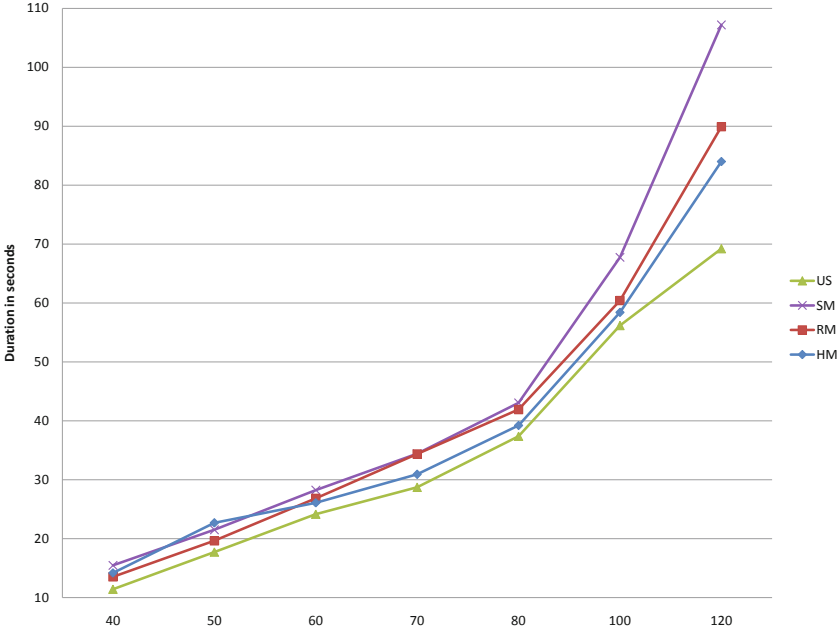


Fig. 4. The average payment processing time wrt. user load

Table 1 shows that one effective way of substantially reducing overhead is by employing additional resources at the remote-side and shift all verification to the auxiliary side (RM). Figures however show that, at a load of 120 users, overhead spike even with this monitoring strategy reaching a level of 30% overhead increase. One possible explanation for this is that the communication channel between the system-side and remote-side becomes saturated causing a bottle-neck in the verification operations.

Our proposal towards solving this problem is to have a Hybrid setup, HR, leveraging parts of the verification on the system-side. Figures in Table 1 show that at low user request loads, *e.g.*, 40 and 50 users, RM performs better than a hybrid approach because more of a less-scarce resource *i.e.*, the communication channel, is being used as opposed to CPU usage at the system-side. However, at higher user loads such, *e.g.*, 80, 100 and 120 users, the balance tips in favour of shifting some verification on the system-side, *i.e.*, HR, where the overheads are consistently less than in the case of RM; at these levels, a hybrid approach manages to approximately *half* the overheads introduced by an extreme remote-side monitoring strategy. In more realistic distributions where the level of untrusted (greylisted) users is even lower, a hybrid approach yielded even better results. We conducted further experiments (see Table 3) where despite user load increases, the number of greylisted users was fixed at 14 users. The results yield more significant gains as the number of users increase.

In conclusion, the CPU figures obtained in Table 2 for RM and HM at low user-request loads deserve further comment, since they appear to suggest that introducing monitors sometimes actually reduces CPU usage. This might be attributed to reduced context switching due to the blocked users waiting for monitor feedback.

Table 3. APT per number of users (in secs) with increasing whitelisted users

| Setup | 40 | 70 | 100 | 120 |
|-------|------------|------------|-----------|------------|
| US | 11.4 | 28.7 | 56.2 | 69.2 |
| SM | 15.5 (35%) | 35.2 (22%) | 62.0 (9%) | 99.2 (43%) |
| RM | 13.5 (18%) | 39.0 (36%) | 60.9 (8%) | 72.3 (4%) |
| HM | 14.2 (24%) | 34.3 (19%) | 56.6 (1%) | 69.6 (1%) |

One aspect which is hidden in this quantitative analysis is the fact that the hybrid approach allowed for the localisation of code which goes more naturally on the system side e.g., code accessing data that is already computed on the system side. By contrast, in the remote monitoring approach one would have no option but to duplicate this computation and the associated data, introducing computation redundancy and additional space overheads. In fact, in cases where resources replication is either not feasible or undesirable,⁸ a hybrid approach turns out to be the only viable solution between these two alternatives.

5 Related Work

Optimisation techniques for synchronous monitoring is a key issue in runtime verification. These techniques broadly fall under two main categories: event sampling techniques and static/dynamic analysis. In the first category [116] only a subset of the system events generated are checked by the monitor, typically in line with some periodic overhead upper limit; this arrangement allows the verification instrumentation to give certain guarantees with respect to monitor overheads, at the expense of monitoring precision. In the second category, static analysis is performed on the monitored properties and their instrumentation [413] in order to optimise their footprint. The first class of techniques are not directly applicable to the security-critical systems discussed in the Introduction since certain violations may go undetected. However a degenerate case of sampling may be used in real world instantiations of our approach acting as a method of last resort when the verification overheads overburden the system. The second class of techniques are complementary to our approach since the enhanced control over where to instrument monitors gives further scope for static analysis to optimise such placements.

To the best of our knowledge, Java-MaC [8] is the only runtime verification tool that implicitly places a boundary between the system and the verifier (albeit with no support for flexibility), by distributing verification across nodes and potentially lowering monitoring overheads. The monitor can however only be located on the verifier side; we argued earlier why this placement strategy may not always yield an optimal level of overheads. Other tools such as JavaMOP [9] and LARVA [5] can support our proposed architecture *indirectly*, since they allow full Java expressivity for monitoring checks and actions. This permits monitor instrumentation to use monitoring actions to open connections and instruct remote deployment of verification checks. However, such an arrangement is far from ideal as it complicates immensely the specification of properties:

⁸ Issues such as data privacy may prove to be one such stumbling block.

it requires additional knowledge of Java distribution mechanisms, thereby discouraging the adoption of our proposed architecture. Moreover, this indirect approach clutters the monitoring code which, in turn, makes monitoring more error prone.

6 Conclusions and Future Work

We have proposed a novel runtime verification technique facilitating the engineering of runtime monitoring over highly parallel systems, thus minimising the inherent overheads introduced by the verification process. By elevating a configurable monitoring boundary to the specification level, the technique allows the user to offload computationally expensive verification checks to a remote site while leveraging the added communication overhead by keeping lightweight verification checks at the site where the monitored system is executing.

The technique has been implemented as part of a runtime monitoring tool called polyLARVA, which takes guarded-command style specification scripts and automatically synthesises the system instrumentation together with the respective monitor verifying the script. The tool allows the user to specify the location of where conditions and actions are to be executed; these delineations correspond to configurable monitoring boundary of the technique and give control over how system resources are managed.

We have also shown how our approach enables alternative monitor configurations that lower overheads through tests performed on an online portal handling multiple user requests in parallel. Our results indicate that different overhead savings can be obtained under different monitoring boundary specifications, depending on the level of service load experienced by the system and on whether monitoring is processing or communication intensive. These results show that while resource-intensive monitoring benefits substantially from remote monitoring, communication does not scale up as well as other resources. Balancing resources overheads against the communication incurred turned out to be essential to lower these overheads, and polyLARVA facilitated the necessary fine-tuning immensely.

Future Work: We plan to extend our work in various ways. We intend to further our tests to deployment architectures involving more than one node for the remote-side of the monitoring boundary. We are also exploring ways how to integrate our optimisation technique with complementary techniques such as sampling. These efforts should yield even lower monitoring overheads which would increase the appeal of the technique to real-world scenarios with more stringent performance requirements, as opposed to security-critical systems.

Work is already underway to extend polyLARVA so that it can handle monitoring of systems that are developed using *different technologies and languages*, thus broadening the appeal of the tool.⁹ To this end, our present monitoring boundary implementation over TCP should facilitate technology-agnostic monitoring of multi-technology systems.

The elevation of the monitoring boundary at the specification level lends itself to further conceptual development. We plan to extend our technique to handle *dynamic*

⁹ At the time of writing, polyLARVA supports monitoring of systems written in Java.

monitor partitioning across the system/remote-sides that can reconfigure itself as the system evolves, thus adapting to changes such as fluctuating system loads. The monitoring boundary also gives scope for various static analyses that can be carried out on our existing polyLARVA scripts so as to obtain *automated* partitioning and placement of monitors across the boundary.

References

1. Arnold, M., Vechev, M., Yahav, E.: Qvm: an efficient runtime for detecting defects in deployed systems. SIGPLAN Not. 43, 143–162 (2008)
2. Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.): RV 2010. LNCS, vol. 6418. Springer, Heidelberg (2010)
3. Bodden, E., Chen, F., Rosu, G.: Dependent advice: a general approach to optimizing history-based aspects. In: Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development, AOSD 2009, pp. 3–14. ACM (2009)
4. Bodden, E., Hendren, L., Lhoták, O.: A Staged Static Program Analysis to Improve the Performance of Runtime Monitoring. In: Bateni, M. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 525–549. Springer, Heidelberg (2007)
5. Colombo, C., Pace, G.J., Schneider, G.: Larva — safer monitoring of real-time java programs (tool paper). In: Seventh IEEE International Conference on Software Engineering and Formal Methods (SEFM), pp. 33–37. IEEE (2009)
6. Dwyer, M.B., Diep, M., Elbaum, S.: Reducing the cost of path property monitoring through sampling. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE 2008, pp. 228–237. IEEE (2008)
7. Geilen, M.: On the construction of monitors for temporal logic properties. Electr. Notes Theor. Comput. Sci. 55(2), 181–199 (2001)
8. Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-mac: A run-time assurance approach for java programs. Formal Methods in System Design 24, 129–155 (2004)
9. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. International Journal on Software Techniques for Technology Transfer (2011) (to appear)
10. Sen, K., Roşu, G., Agha, G.: Generating Optimal Linear Temporal Logic Monitors by Coinduction. In: Saraswat, V.A. (ed.) ASIAN 2003. LNCS, vol. 2896, pp. 260–275. Springer, Heidelberg (2003)

Frama-C

A Software Analysis Perspective*

Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto,
Julien Signoles, and Boris Yakobowski

with Patrick Baudin, Richard Bonichon, Bernard Botella, Loïc Correnson,
Zaynah Dargaye, Philippe Herrmann, Benjamin Monate, Yannick Moy,
Anne Pacalet, Armand Puccetti, Muriel Roger, and Nicky Williams

CEA, LIST, Software Safety Laboratory, PC 174, 91191 Gif-sur-Yvette France
{Pascal.Cuoq,Florent.Kirchner,Nikolai.Kosmatov,
Virgile.Prevosto,Julien.Signoles,Boris.Yakobowski}@cea.fr

Abstract. Frama-C is a source code analysis platform that aims at conducting verification of industrial-size C programs. It provides its users with a collection of plug-ins that perform static analysis, deductive verification, and testing, for safety- and security-critical software. Collaborative verification across cooperating plug-ins is enabled by their integration on top of a shared kernel and datastructures, and their compliance to a common specification language. This foundational article presents a consolidated view of the platform, its main and composite analyses, and some of its industrial achievements.

1 Introduction

The past forty years have seen much of the groundwork of formal software analysis being laid. Several angles and theoretical avenues have been explored, from deductive reasoning to abstract interpretation to program transformation to concolic testing. While much remains to be done from an academic standpoint, some of the major advances in these fields are already being successfully implemented [18,41,25,46,51] – and met with growing industrial interest. The ensuing push for mainstream diffusion of software analysis techniques has raised several challenges. Chief among them are: *a.* their scalability, *b.* their interoperability, and *c.* the soundness of their results.

Point [7](#) is predictably important from the point of view of adoptability. Scaling to large problems is a prerequisite for the industrial diffusion of software analysis and verification techniques. It also represents a mean to better understand how language idioms (e.g. pointers, unions, or dynamic memory allocation) influence the underlying architecture of large software developments. Overall, achieving scalability in the design of software analyzers for a wide range of software patterns remains a difficult question.

* This work was partly supported by ANR U3CAT and Veridyc, and FUI9 Hi-Lite projects.

Point [\[6\]](#) – interoperability – enables the design of elaborate program analyses. Consider indeed the interplay between program analyses and transformations [\[22\]](#), the complementarity of forward and backward analyses [\[2\]](#), or the precision gain afforded when combining static and dynamic approaches [\[4\]](#). Yet running multiple source code analyses and synthesizing their results in a coherent fashion requires carefully thought-out mechanisms.

Point [\[7\]](#) – soundness – is a strong differentiator for formal approaches. By using tools that *over-approximate* all program behaviors, industrial users are assured that none of the errors they are looking for remain undetected. This guarantee stands in stark contrast with the bug-finding capabilities of heuristic analyzers, and is paramount in the evaluation of critical software. But the design and implementation costs of such high-integrity solutions are hard to expend.

The Frama-C software analysis platform provides a collection of scalable, interoperable, and sound software analyses for the industrial analysis of ISO C99 source code. The platform is based on a common *kernel*, which hosts analyzers as collaborating *plug-ins* and uses the ACSL formal specification language as a *lingua franca*. Frama-C includes three fundamental plug-ins based on abstract interpretation, deductive verification, and concolic testing; and a series of derived plug-ins which build elaborate analyses upon the former. In addition, the extensibility of the overall platform, and its open-source licensing, have fostered the development of an ecosystem of independent third-party plug-ins. This article is intended as a foundational reference to the platform, its three main analyses, and its most salient derived plug-ins.

2 The Platform Kernel

2.1 Architecture

Figure [\[1\]](#) shows a functional view of the Frama-C architecture. Frama-C is based on CIL [\[43\]](#), a front-end for C that parses ISO C99 programs into their normalized representation: loop constructs are given a single form, expressions have no side-effects, etc. Frama-C extends CIL to support dedicated source code annotations expressed in ACSL (see § [2.2](#)). This modified CIL front-end produces the C + ACSL AST, an abstract view of the program shared among all analyzers. This AST takes into account machine-dependent parameters (size of integral types, endianness, *etc*) which can easily be modified by the end-user.

The Frama-C kernel provides several services, helping plug-in development [\[49\]](#) and providing convenient features to the end-user.

- Messages, source code and annotations are uniformly displayed; parameters and command line options are homogeneously handled.
- A journal of user actions can be synthesized, and be replayed afterwards, a feature of interest in debugging and qualification contexts.
- A project system, presented in § [2.3](#), isolates unrelated program representations, and guarantees the integrity of their analyses.
- Consistency mechanisms control the collaboration between analyzers (§ [2.4](#)).

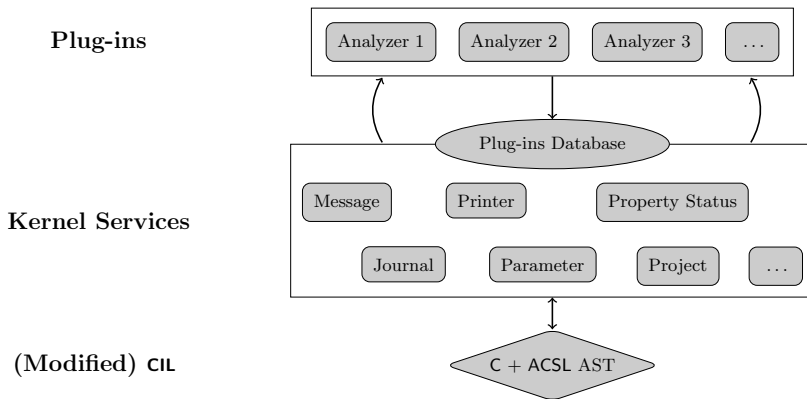


Fig. 1. Frama-C's Functional View

Analzers are developed as separate plug-ins on top of the kernel. Plug-ins are dynamically linked against the kernel to offer new analyses, or to modify existing ones. Any plug-in can register new services in a *plug-ins database* stored in the kernel, making these services available to all plug-ins.

2.2 ACSL

Functional properties of C programs can be expressed within Frama-C as ACSL annotations [3]. ACSL is a formal specification language inspired by Java's JML [9], both being based on the notion of function contract introduced by Eiffel [42]. In effect, the specification of a function states the pre-conditions it **requires** from its caller and the post-conditions it **ensures** when returning. Among these post-conditions, one kind of clause plays a particular role by saying which memory locations the function **assigns**, i.e. which locations might have a different value between the pre- and the post-state.

For instance, Fig. 2 provides a specification for a `swap` function. The first pre-condition states that the two arguments must be valid (`int`) pointers, i.e. that dereferencing `a` or `b` will not produce a run-time error. In addition, the second pre-condition asks that the two locations do not overlap. `\valid` and `\separated` are two built-in predicates: ACSL features various functions and predicates to describe memory states. However, it does not introduce any notion beyond the C standard, leaving each plug-in free to perform its own abstractions over the concrete memory layout. The `assigns` clause states that only the locations pointed

```

1 /*@ requires \valid(a) && \valid(b); requires \separated(a,b);
2   assigns *a, *b;
3   ensures *a == \at(*b,Pre) && *b == \at(*a,Pre); */
4 void swap(int* a, int* b);

```

Fig. 2. Example of ACSL specification

to by `a` and `b` might be modified by a call to `swap`; any other memory location is untouched. Finally, the post-condition says that at the end of the function, `*a` contains the value that was in `*b` in the pre-state, and *vice versa*.

In addition to function specifications, ACSL offers the possibility of writing annotations in the code, in the form of **assertions** (properties that must be true at a given point) or **loop invariants** (properties that must be preserved across any number of loop steps). Annotations are written in first-order logic, and it is possible to define custom functions and predicates for use in annotations together with ACSL built-ins. Plug-ins can provide a validity status to any ACSL property and generate ACSL annotations. This allows ACSL annotations to play an important role in the communication between plug-ins, as explicated in § 2.4

2.3 Projects

Frama-C allows a user to work on several programs in parallel thanks to the notion of *project*. A project consistently stores a program with all its required information, including results computed by analyzers and their parameters. Several projects may coexist in memory at the same time. A non-interference theorem guarantees project partitioning [48]: any modification on a value of a project \mathcal{P} does not impact a value of another project \mathcal{P}' .

Such a feature is of particular interest when a program transformer like Slicing (§ 6.1) or Aorai (§ 6.2) is used. The result of the transformation is a fresh AST that coexists with the original, making backtracking and comparisons easy. Another use of projects is to process the same program in different ways – for instance with different analysis parameters.

2.4 Analyzers Collaboration

In Frama-C, analyzers can collaborate in two different ways: either *sequentially*, by chaining analysis results to perform complex operations; or *in parallel*, by combining partial analysis results into a full program verification.

The former consists in using the results of an analyzer as input to another one thanks to the plug-ins database stored by the Frama-C kernel. Refer to § 6.1 for a comprehensive illustration of a sequential analysis.

The parallel collaboration of analyzers consists in verifying a program by heterogeneous means. ACSL is used to this end as a collaborative language: plug-ins generate program annotations, which are then validated by other plug-ins. Partial results coming from various plug-ins are integrated by the kernel to provide a consolidated status of the validity of all ACSL properties. For instance, when the Value plug-in (§ 3) is unable to ensure the validity of a pointer p , it emits an unproved ACSL annotation `assert !valid(p)`. In accordance with the underlying blocking semantics, it assumes that p is valid from this program point onwards. The WP plug-in (§ 4) may later be used to lift this hypothesis. The kernel automatically computes the validity status of each program property from the information provided by all analyzers and ensures the consistency of the entire verification process [16]: “if the consolidated status of a property is computed as

valid [resp. invalid] by the kernel, then the property is valid [resp. invalid] with respect to ACSL's semantics".

3 Fundamental Analysis: Abstract Interpretation

The `Value` plugin (short for Value Analysis) is a forward dataflow analysis based on the principles of *abstract interpretation* [17]. Abstract interpretation links a *concrete* semantics, typically the set of all possible executions of a program, to a more coarse-grained, *abstract* one. Any transformation in the concrete semantics must have an abstract counterpart that captures all possible outcomes of the concrete operation. This ensures that the abstract semantics is a sound approximation of the runtime behavior of the program.

`Value`, and abstract interpreters in general, proceed by symbolic execution of the program, translating all operations into the abstract semantics. Termination of looping constructs is ensured by *widening* operations. For function calls, `Value` proceeds essentially by recursive inlining of the function (recursive functions are currently not handled). This ensures that the analysis is fully context-sensitive. If needed, the user can abstract overly complex functions by an ACSL contract, verified by hand or discharged with another analysis.

Abstract Domains. The domains currently used by `Value` to represent the abstract semantics are described below.

Integer Computations. Small sets of integers are represented as sets, whereas large sets are represented as intervals with congruence information [30]. For instance, $x \in [3..255], 3\%4$ means that x is such that $3 \leq x \leq 255$ and $x \equiv 3 \pmod{4}$.

Floating-Point Computations. The results of floating-point computations are represented as IEEE 754 [34] double-precision finite intervals. Operations on single-precision floats are stored as doubles, but are rounded as necessary. Obtaining infinities or NaN is treated as undesirable errors.

Pointers and Memory. To verify that invalid (e.g. out-of-bounds) array/pointer accesses cannot occur in the target program, `Value` assumes that the program does not purposely use buffer overflows to access neighboring variables [35, §6.5.6:8]. Abstract representation of memory states in a C program reflects this assumption: addresses are seen as offsets with respect to symbolic *base addresses*, and have no relation with actual locations in virtual memory space during execution.

Memory representation is untyped. It is thus straightforward to handle unions and heterogeneous pointer conversions during abstract interpretation. The abstract memory state maps each base address to a representation of a chunk of linear memory. Each such object itself maps ranges of bits to values. Given an array of 32-bit integers τ , and reading from $*((\text{char}*)\tau + 5)$, the analyzer determines that the relevant abstract value is to be found between bits $[40..47]$ of the value bound to $\&\tau$. Using bit as unit, instead of byte, allows to handle

bit-fields [35, §6.2.6.1:4] by fixing a layout strategy (the C standard itself does not specify bit-fields layout, but then again, no more than any kind of data).

Finally, the content of some memory locations is deemed *indeterminate* by the C standard. Examples include uninitialized local variables, struct padding, and dereferencing pointers to variables outside their scope [35, §6.2.4:2]. Having indeterminate contents in memory is not an error, but accessing an indeterminate memory location is. To detect those, the values used to represent the contents of memory locations are taken, not directly from the abstract domain used for the values of an expression, but from the lattice product of this domain with two two-valued domains, one for initializedness and the other for danglingness.

Propagation of Unjoined States. Value’s domains are non-relational. Instead, the datastructures representing the abstract semantics have been heavily optimized for speed and reduced memory footprint, to allow the independent propagation of multiple distinct states per statement. This alleviates for a large part the need for relational domains, by implicitly encoding relations in the disjunction of abstract states. Typically, by choosing to propagate k distinct states, the user can ensure that simple loops with less than k iterations are entirely unrolled. Successive conditionals are also handled more precisely: the abstract states remain separate after the two branches of the conditional have been analyzed.

Alarms. Each time a statement is analyzed, any operation that can lead to an undefined behavior (e.g. division by zero, out-of-bounds access, etc.) is checked, typically by verifying the range of the involved expression – the denominator of the division, the index of the array access, etc.

When the abstract semantics guarantees that no undesirable value can occur, one obtains a static guarantee that the operation always executes safely. Otherwise, Value reports the possible error by an *alarm*, expressed as an ACSL assertion. This alarm may signal a real error if the operation fails at runtime on at least one execution, or a *false alarm*, caused by the difference in precision between the concrete and abstract semantics. More precise state propagation typically results in fewer false alarms, but lengthen analysis time. Upon emitting an alarm, the analyzer reduces the propagated state accordingly, and proceeds onwards.

4 Fundamental Analysis: Deductive Verification

The WP plug-in is named after the *Weakest Precondition* calculus, a technique used to prove program properties initiated by Hoare [32], Floyd [28] and Dijkstra [24]. Recent tools implement this technique efficiently, for instance Boogie [38] and Why [27]. Jessie [39], a Frama-C plug-in developed at INRIA, also implements this technique for C by compiling programs into the Why language. Frama-C’s WP plug-in is a novel implementation of a Weakest Precondition calculus for generating verification conditions (VC) for C programs with ACSL annotations. It differs from other implementations in two respects. First, WP focuses on parametrization with respect to the memory model. Second, it aims at being

fully integrated into the Frama-C platform, and to ease collaboration with the other verification plug-ins (especially Value) as outlined in § 2.4.

The choice of a memory model is a key ingredient of Hoare logic-based VC generators that target C programs (or more generally a language with memory references). A weakest precondition calculus operates over a language that only manipulates plain variables. In order to account for pointers, memory accesses (both for reading and writing) must be represented in the underlying logic. The simplest representation uses a single functional array for the whole memory. However, this has a drawback: any update to the array (the representation of an assignment $*p=v$) has a potential impact on the whole memory – any variable might have been modified. In practice, proof obligations quickly become intractable. Thus, various refinements have been proposed, in particular by Bornat [6], building upon earlier work by Burstall [10]. The idea of such memory models is to use distinct arrays to represent parts of the memory known to be separated, e.g. distinct fields of the same structure in the “component-as-array trick” of Burstall and Bornat. In this setting, an update to one of the arrays will not affect the properties of the others, leading to more manageable VC.

However, abstract memory models sometimes restrict the functions that can be analyzed. Indeed, a given model can only be used to verify code that does not create aliases between pointers that are considered *a priori* separated by the model. In particular, Burstall-Bornat models that rely on static type information to partition the memory are not able to cope with programs that use pointer casts or some form of `union` types.

In order to generate simpler VC when possible while still being able to verify low-level programs, WP provides different memory models that the user can choose for each ACSL property. The current version offers three main models:

- The most abstract model, `hoare`, roughly corresponds to Caveat’s model [46]. It can only be used over functions that do not explicitly assign pointers or take the address of a variable, but provides compact VC.
- The default model is `store`. It is a classical Burstall-Bornat model, that supports pointer aliasing, but neither cast nor union types.
- The runtime model is designed for code that perform low-level memory operations. In this model, the memory is seen as a single array of `char`, so that most C operations can be taken into account, at the expense of the complexity of the generated VC.

As a refinement, `store` and `runtime` can avoid converting assignments into array updates when the code falls in the subset supported by `hoare`. In particular, variables whose addresses are not taken and plain references – pointers that are neither assigned nor used in a pointer arithmetic operation – are translated as standard Hoare logic variables. This way, the overhead of other models with respect to `hoare` is kept to the places where it is really needed.

Once a VC has been generated, it must be discharged. WP natively supports two theorem provers: the automated SMT solver Alt-ergo [14], and the Coq proof assistant [15]. Other automated provers can also be used through the multi-prover backend of Why. Advantages of using a dedicated back-end rather than

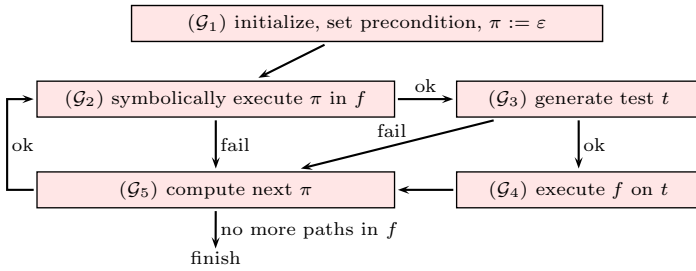


Fig. 3. The PathCrawler test generation method

relying completely on Why are twofold. First, it removes a dependency over an external tool, meaning that for verification of critical software, there is one component less that needs to be assessed. Second, WP can take advantage of specific features of Alt-ergo, most notably native support for arrays and records (that occur quite often in typical VC), that are not supported by Why yet.

In contrast to Jessie, that relies on an external tool for VC generation, WP operates entirely within Frama-C. In particular, WP fills the property status table described in § 2.4 for each annotation on which it is run. The dependencies of such a status are the annotations taken as hypothesis during the weakest-precondition calculus, the memory model that has been used, and the theorem prover that ultimately discharged the VC. The memory model has a direct impact on the validity of the result: an annotation can very well be valid under model store but not under runtime, as the former entails implicit separation hypotheses that are not present in the latter. In theory, the choice of a theorem prover is not relevant for the correctness of the status, but this information is important to fully determine a trusted toolchain.

Having WP properly embedded into Frama-C also allows for a fine-grained control over the annotations one wants to verify with the plug-in. WP provides the necessary interface at all levels (command-line option, programmatic API, and GUI) to verify targetted annotations (e.g. those yet unverified by other means in Frama-C, cf. § 2.4) as well as to generate all the VC related to a C function.

5 Fundamental Analysis: Concolic Testing

Given a C program p under test and a precondition restricting its inputs, the PathCrawler plug-in generates test cases respecting various test coverage criteria. The *all-path* criterion requires covering all feasible program paths of p . Since the exhaustive exploration of all paths is usually impossible for real-life programs, the *k-path* criterion restricts exploration to paths with at most k consecutive iterations of each loop. The PathCrawler [51,7] method for test generation is similar to the so-called *concolic* (*concrete+symbolic*) approach and to Dynamic Symbolic Execution (DSE), implemented by other tools (e.g. DART, CUTE, PEX, SAGE, KLEE).

PathCrawler starts by: *a.* constructing an instrumented version of p that will trace the program path exercised by the execution of a test case, and *b.* generating the constraints which represent the semantics of each instruction in p . The next step, illustrated by Fig. 3, is the generation and resolution of constraint systems to produce the test cases for a set of paths Π that satisfy the coverage criterion. This is done in the ECLiPSe Prolog environment [47] and uses Constraint Logic Programming. Given a *path prefix* π , i.e. a partial program path in p , the main idea [37] is to solve the constraints corresponding to the symbolic execution of p along π . A constraint store is maintained during resolution, and aggregates the various constraints encountered during the symbolic execution of π . The test generation method follows the following steps:

- (\mathcal{G}_1) Create a logical variable for each input. Add constraints for the precondition into the constraint store. Let the initial path prefix π be empty (i.e. the first test case can be any test case satisfying the precondition). Continue to Step (\mathcal{G}_2).
- (\mathcal{G}_2) Symbolically execute the path π : add constraints and update the memory according to the instructions in π . If some constraint fails, continue to Step (\mathcal{G}_5). Otherwise, continue to Step (\mathcal{G}_3).
- (\mathcal{G}_3) Call the constraint solver to generate a test case t , that is, concrete values for the inputs, satisfying the current constraints. If it fails, go to Step (\mathcal{G}_5). Otherwise, continue to Step (\mathcal{G}_4).
- (\mathcal{G}_4) Run a traced execution of the program on the test case t generated in the previous step to obtain the complete execution path. The complete path must start by π . Continue to Step (\mathcal{G}_5).
- (\mathcal{G}_5) Compute the next partial path, π , to cover. π is constructed by “taking another branch” in one of the complete paths already covered by a previous test case. This ensures that all feasible paths are covered (as long as the constraint solver can find a solution in a reasonable time) and that only the shortest infeasible prefix of each infeasible path is explored.

PathCrawler uses Colibri, a specialized constraint solving library developed at CEA LIST and shared with such other testing tools as GATeL [40] and OSMOSE [1]. Colibri provides a variety of types and constraints (including non-linear constraints), primitives for labelling procedures, support for floating point numbers and efficient constraint resolution. PathCrawler is a proprietary plug-in, also available in the form of a freely accessible test-case generation web service [36].

6 Derived Analyses

6.1 Distilling Values

The outputs of the Value plugin are twofold. In addition to emitting alarms for statements it cannot guarantee are safe (§3), Value automatically computes a per-statement over-approximation of the possible values for all memory locations. The derived analyses below reuse those synthetic results. In each case, the analysis is sound. Value’s results are used to evaluate array indexes or resolve pointers, ensuring that e.g. pointer aliasing are always detected.

Outputs: over-approximates the locations a function may write to.

Operational inputs: over-approximates the locations whose initial values are used by the function.

Functional dependencies: computes a relation between outputs and inputs of a function; x FROM y , $t[1]$ (and SELF) means that output x is either unchanged (SELF), or that its new value can be computed exclusively from inputs y and $t[1]$.

Program Dependency Graph (PDG): produces an intra-procedural graph that expresses the *data* and *control* dependencies between the instructions of a function, used as a stepping stone for various analyzes [26].

Defs: over-approximates which statements define a given memory location.

Impact: computes the values and statements impacted (directly or transitively) by the side effects of a chosen statement.

Slicing: returns a reduced program (a *slice*), equivalent to the original program according to a given *slicing criterion* [33]. Possible criteria include preserving a given statement, all calls to a function, a given alarm, etc.

Analyses such as Defs or Impact make compelling code understanding tools, as they express in a very concise way the relationships between various parts of a program. Slicing goes one step further: while it is essentially dual to the impact analysis, it also builds reduced, self-contained programs, that can be re-analyzed independently. Those three analyses are fully inter-procedural.

The analyses above illustrate sequential collaboration (§ 2.4). PDG makes heavy use of Functional dependencies, while Defs, Impact and Slicing leverage the information given by PDG. Some of those analyses can optionally compute *callwise* versions of their results, yielding one result per syntactic call, instead of one result per function. The improved precision automatically benefits the derived analyses. All the results are stored by the Frama-C kernel, and can be reused without being recomputed.

6.2 Annotation Generator

The Aoraï plug-in [50,31] plays a particular role among the core Frama-C plug-ins. Indeed, it is one of the few whose primary aim is to generate ACSL annotations rather than attempting to verify them. Aoraï provides a way to specify that all possible executions of a program respect a given sequence of events, namely the call and return of functions, possibly with constraints on the program's state at each event. The specification itself can be given either as a Linear Temporal Logic (LTL, [45]) formula or in the form of an automaton. In the former case, Aoraï uses ltl2ba [29] to obtain an equivalent Büchi automaton.

Given an automaton, Aoraï provides ACSL specifications for each function ε in the original C code. This instrumentation, summarized in Fig. 4, consists of two main parts: two prototypes whose specification represents the transitions for an atomic event (call or return from ε), and the specification of ε itself. As the automaton is not always deterministic, Aoraï represents active states by a set of boolean variables (`aorai_state_*`). Functions `advance_automaton_*` provide for each such variable a complete set of `behaviors` indicating when they are set to 1 or 0.

```

1  /*@ behavior transition_1: assumes aorai_state_S_0 == 1 && condition;
2     ensures aorai_state_S_next == 1; ... */
3  void advance_automaton_call_f(int x);
4
5  /*@ requires aorai_state_S_0 == 1 || aorai_state_S_1 == 1 || ...;
6     requires aorai_state_S_0 == 1 ==> has_possible_transition_S0; ...
7     ensures aorai_state_S_2 == 1 || aorai_state_S_3 == 1 || ...;
8     ensures \old(aorai_state_S_0 == 1) ==> aorai_state_S_2 == 1 || ...; ...
9     ensures aorai_state_S_2 == 1 ==> program_state_when_in_S_2; ... */
10 int f(int x) {
11     advance_automaton_call_f(x);
12     // Body of f
13     advance_automaton_return_f(result);
14     return result; }

```

Fig. 4. Aoraï’s instrumentation

The specification of ε comprises various items. First, at least one state among a given set must be active before the call. This set is determined by a rough static analysis made by Aoraï beforehand. In addition, for each active state, at least one transition must be activated by the call event. The main post-condition is that when the function returns, at least one state is active among those deemed possible by Aoraï’s static analysis. It is refined by additional clauses relating active initial states with active final states, and the state of the program itself with the active final states. Finally the `main` function has an additional post-condition stating that at least one of the acceptance states must be active at the end of the function (Aoraï does not consider infinite programs at the moment, *i.e.* it can only check for safety properties and cannot be used for liveness).

Annotation generation is geared towards the use of deductive verification plugins such as WP and Jessie for the verification of the specification. In particular, the refined post-conditions are mainly useful for propagating information to the callers of ε in an Hoare-logic based setting. Likewise Aoraï also generates loop invariants for the same purpose. However it does not preclude the use of the Value plug-in to validate its specification, and therefore attempts to generate annotations that fit in the subset of ACSL that is understood by Value.

6.3 SANTE

The Sante Frama-C plug-in (Static ANalysis and TEsting) [13] enhances static analysis results by testing. Given a C program p with a precondition, it detects possible runtime errors (currently divisions by zero and out-of-bound array accesses) in p and aims to classify them as real bugs or false alarms.

The Sante method contains three main steps illustrated in Fig. 5. Sante first calls Value to analyze p and to generate an alarm for each potentially unsafe statement. Next, Slicing is used to reduce the whole program with respect to one or several alarms. It produces one or several slices p_1, p_2, \dots, p_n . Then, for each p_i , PathCrawler explores program paths and tries to generate test cases confirming the alarms present in p_i . If a test case activating an alarm is found, the alarm is confirmed and classified as a bug. If all feasible paths were explored for some

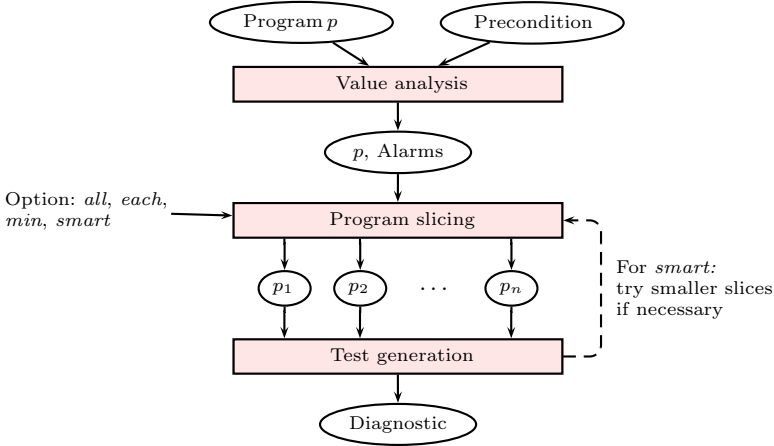


Fig. 5. Overview of the Sante method

slice p_i , all unconfirmed alarms in p_i are classified *safe*, i.e. they are in fact false alarms. If PathCrawler was used with a partial criterion (k -path), or stopped by a timeout before finishing the exploration of all paths of p_i , Sante cannot conclude and the statuses of unconfirmed alarms in p_i remain unknown.

The number of slices generated, hence the number of test generation sessions, is influenced by various Sante options. The `all` option generates a unique slice p_1 including all alarms of p , while the `each` option generates a slice for each alarm. Options `min` and `smart` take advantage of alarm dependencies (as computed by the dependency analysis) to slice related alarms together. The `smart` option improves `min` by iteratively refining the slices as long as one can hope to classify more alarms running PathCrawler on a smaller slice. Initial experiments with Sante are available in [13].

7 Adoption

Adoption in the academic world has stemmed from a variety of partnerships. Foremost is the Jessie plug-in [39] developed at Inria, which relies on a separation memory model but whose internal representation precludes its combination with other plug-ins. Verimag researchers have implemented a taint analysis [11], producing explicit dependency chains pondered by risk quantifiers. Demay *et al* generate security monitors based on fine-grained feedback from the Value plug-in [23]. Berthome *et al* [5] propose a source-code model for verifying physical attacks on smart cards, and use Value to verify it. Bouajjani *et al* [8] automatically synthesize invariants of sequential programs with singly-linked lists. Finally, although the variety of objectives a static analyzer can have, and the variety of design choices for a given objective, make it difficult to benchmark static

analyzers, Chatzieftheriou and Katsaros [12] have valiantly produced one such comparison including Frama-C's Value plug-in.

On the industrial side, companies are adopting or evaluating Frama-C. Delmas *et al* verify the compliance to domain-specific coding standards [20]; their plug-in is undergoing deployment and qualification. At the same company, the value analysis is used to verify the control and data flows of a DAL C, 40-kloc ARINC 653 application [21]. Pariente and Ledinot [44] verify flight control system code using a combination of Frama-C plugins, including Value and Slicing. Their contribution includes a favorable evaluation of the cost-effectiveness of their adoption compared to traditional verification techniques. Yakobowski *et al* use Value in collaboration with WP to check the absence of runtime errors in a 50 kloc instrumentation and control (I&C) nuclear code [52]. Through these successes and over the past few years, Frama-C has demonstrated its adoptability within diverse industrial environments.

8 Conclusion

This article attempts to distill a unified presentation of the Frama-C platform from a software analysis perspective. Frama-C answers the combined introductory challenges of scalability, interoperability, and soundness with a unique architecture and a robust set of analyzers. Its core set of tools and functionalities – about 150 kloc. developed over the span of 7 years [19] – has given rise to a flourishing ecosystem of software analyzers. In addition to industrial achievements and partnerships, a community of users and developers has grown and strived, contributing to the dissemination of the tools. This growth, fostered by a number of active communication channels¹, should be interpreted as a testimony to the health of the software analysis community, and good omens for its future.

References

1. Bardin, S., Herrmann, P.: OSMOSE: automatic structural testing of executables. *Software Testing, Verification and Reliability* 21(1), 29–54 (2011)
2. Bardin, S., Herrmann, P., Védrine, F.: Refinement-Based CFG Reconstruction from Unstructured Programs. In: Jhala, R., Schmidt, D. (eds.) *VMCAI 2011*. LNCS, vol. 6538, pp. 54–69. Springer, Heidelberg (2011)
3. Baudin, P., et al.: ACSL: ANSI/ISO C Specification Language Preliminary design, version 1.5 (2010), http://frama-c.com/downloads/acsl_1.5.pdf
4. Beckman, N., et al.: Proofs from tests. *IEEE Trans. Software Eng.* 36(4), 495–508 (2010)
5. Berthomé, P., et al.: Attack model for verification of interval security properties for smart card c codes. In: *PLAS*, pp. 2:1–2:12. ACM (2010)
6. Bornat, R.: Proving Pointer Programs in Hoare Logic. In: Backhouse, R., Oliveira, J.N. (eds.) *MPC 2000*. LNCS, vol. 1837, pp. 102–126. Springer, Heidelberg (2000)
7. Botella, B., et al.: Automating structural testing of C programs: Experience with PathCrawler. In: *AST*, pp. 70–78 (2009)

¹ See <http://frama-c.com/support.html>.

8. Bouajjani, A., et al.: On inter-procedural analysis of programs with lists and data. In: PLDI, pp. 578–589 (2011)
9. Burdy, L., et al.: An overview of JML tools and applications. *Software Tools for Technology Transfer* 7(3), 212–232 (2005)
10. Burstall, R.M.: Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence* 7, 23–50 (1972)
11. Ceara, D., Mounier, L., Potet, M.-L.: Taint dependency sequences: A characterization of insecure execution paths based on input-sensitive cause sequences. In: ICSTW, Washington, DC, USA, pp. 371–380 (2010)
12. Chatzieftheriou, G., Katsaros, P.: Test-driving static analysis tools in search of C code vulnerabilities. In: COMPSAC Workshops, pp. 96–103. IEEE (2011)
13. Chebaro, O., et al.: Program slicing enhances a verification technique combining static and dynamic analysis. In: SAC (to appear, 2012)
14. Conchon, S., et al.: The Alt-Ergo Automated Theorem Prover, <http://alt-ergo.lri.fr>
15. Coq Development Team. The Coq Proof Assistant Reference Manual, v8.3 edition (2011), <http://coq.inria.fr/>
16. Correnson, L., Signoles, J.: Combining Analyses for C Program Verification. In: Stoelinga, M., Pinger, R. (eds.) FMICS 2012. LNCS, vol. 7437, pp. 108–130. Springer, Heidelberg (2012)
17. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
18. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTREÉ Analyzer. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005)
19. Cuoq, P., et al.: Experience report: OCaml for an industrial-strength static analysis framework. In: ICFP, pp. 281–286. ACM (2009)
20. Delmas, D., et al.: Taster, a Frama-C plug-in to encode Coding Standards. In: ERTSS (May 2010)
21. Delmas, D., et al.: Fan-C, a Frama-C plug-in for data flow verification. In: ERTSS (2012)
22. Demange, D., Jensen, T., Pichardie, D.: A Provably Correct Stackless Intermediate Representation for Java Bytecode. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 97–113. Springer, Heidelberg (2010)
23. Demay, J.-C., Totel, E., Tronel, F.: Sidan: A tool dedicated to software instrumentation for detecting attacks on non-control-data. In: CRiSIS (October 2009)
24. Dijkstra, E.W.: A constructive approach to program correctness. *BIT Numerical Mathematics*. Springer (1968)
25. Dahlweid, M., et al.: VCC: Contract-based modular verification of concurrent C. In: ICSE Companion. IEEE (2009)
26. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.* 9(3), 319–349 (1987)
27. Filliâtre, J.-C.: Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud (March 2003)
28. Floyd, R.W.: Assigning meanings to programs. *Proceedings of the American Mathematical Society Symposia on Applied Mathematics* 19 (1967)
29. Gastin, P., Oddoux, D.: Fast LTL to Büchi Automata Translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001)

30. Granger, P.: Static Analysis of Linear Congruence Equalities Among Variables of a Program. In: Abramsky, S. (ed.) CAAP 1991 and TAPSOFT 1991. LNCS, vol. 493, pp. 169–192. Springer, Heidelberg (1991)
31. Gros Lambert, J., Stouls, N.: Vérification de propriétés LTL sur des programmes C par génération d'annotations. In: AFADL (2009) (in French)
32. Hoare, C.A.R.: An axiomatic basis for computer programming. *Communications of the ACM* 12(10) (1969)
33. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. *PLDI, SIGPLAN Notices* 23(7), 35–46 (1988)
34. IEEE Std 754-2008. IEEE standard for floating-point arithmetic. Technical report (2008), <http://dx.doi.org/10.1109/IEEESTD.2008.4610935>
35. ISO/IEC JTC1/SC22/WG14. 9899:TC3: Programming Languages—C (2007), <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf>
36. Kosmatov, N.: Online version of PathCrawler, <http://pathcrawler-online.com/>
37. Kosmatov, N.: XI: Constraint-Based Techniques for Software Testing. In: Artificial Intelligence Applications for Improved Software Engineering Development: New Prospects, IGI Global (2010)
38. Leino, K.R.M.: This is Boogie 2. Microsoft Research (2008)
39. Marché, C., Moy, Y.: The Jessie plugin for Deduction Verification in Frama-C, version 2.30. INRIA (2012), <http://krakatoa.lri.fr/jessie.pdf>
40. Marre, B., Arnould, A.: Test sequences generation from Lustre descriptions: GATeL. In: ASE, Grenoble, France, pp. 229–237 (September 2000)
41. MathWorks. Polyspace, <http://www.mathworks.com/products/polyspace>
42. Meyer, B.: Object-oriented Software Construction. Prentice-Hall (1997)
43. Nacula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: CC 2002. LNCS, vol. 2304, p. 213. Springer, Heidelberg (2002)
44. Pariente, D., Ledinot, E.: Formal Verification of Industrial C Code using Frama-C: a Case Study. In: FoVeOOS (2010)
45. Pnueli, A.: The temporal logic of programs. In: FOCS. IEEE (1977)
46. Randimbivololona, F., Souyris, J., Baudin, P., Pacalet, A., Raguideau, J., Schoen, D.: Applying Formal Proof Techniques to Avionics Software: A Pragmatic Approach. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1709, pp. 1798–1815. Springer, Heidelberg (1999)
47. Schimpf, J., Shen, K.: ECLiPSe - from LP to CLP. *Theory and Practice of Logic Programming* 12(1-2), 127–156 (2011)
48. Signoles, J.: Foncteurs impératifs et composés: la notion de projet dans Frama-C. In: JFLA, *Studia Informatica Universalis*, vol. 7(2) (2009) (in French)
49. Signoles, J., Correnson, L., Prevosto, V.: Frama-C Plug-in Development Guide (October 2011), <http://frama-c.com/download/plugin-developer.pdf>
50. Stouls, N., Prevosto, V.: Aorai plug-in tutorial, version Nitrogen-20111001 (October 2011), <http://frama-c.com/download/frama-c-aorai-manual.pdf>
51. Williams, N., Marre, B., Mouy, P., Roger, M.: PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis. In: Dal Cin, M., Kaâniche, M., Pataricza, A. (eds.) EDCC 2005. LNCS, vol. 3463, pp. 281–292. Springer, Heidelberg (2005)
52. Yakobowski, B., et al.: Formal verification of software important to safety using the Frama-C tool suite. In: NPIC (July 2012)

An Optimization Approach for Effective Formalized fUML Model Checking

Islam Abdelhalim, Steve Schneider, and Helen Treharne

Department of Computing, University of Surrey
{i.abdelhalim,s.schneider,h.treharne}@surrey.ac.uk

Abstract. Automatically formalizing fUML models into CSP is a challenging task. However, checking the generated CSP model using FDR2 is far more challenging. That is because the generated CSP model holds many implementation details inherited from the fUML model, as well as the formalization of the non-trivial fUML inter-object communication mechanism. Using the state space compression techniques available in FDR2 (such as supercompilation and compression functions) is not enough to provide an effective model checking that avoids the state explosion problem. In this paper we introduce a novel approach that makes use of a restricted CSP model (because it follows certain formalization rules) to optimize the generated model. As an application of our approach, we design a framework that works on two levels; the first one provides optimization advice to the modeller, while the second one automatically applies optimization rules which transform the CSP model to a more optimized one with a reduced state space.

Implementing and applying the approach on two large case studies demonstrated the effectiveness of the approach. We also prove that the optimization rules are safe to be applied automatically without eliminating important information from the CSP model.

1 Introduction

Formalizing fUML (Foundational Subset for Executable UML) [1] models to a CSP (Communication Sequential Processes) [2] formal representation, has been previously considered [3,4]. This formalization allows modellers to check certain properties in their fUML model without the need for specialist formal methods knowledge. Our motivation for using fUML as a semi-formal modelling language was its restricted standard and the ability to represent the implementation detail of asynchronous systems. However, the automatically generated CSP model that depends on such an inter-object communication mechanism is usually convoluted. Checking such a model using FDR2 [5] is a very challenging task due to the high possibility of the state space explosion problem, or even the length of time taken to perform the model checking.

Toward a solution to minimize the possibility of that problem, we propose in this work an optimization approach that works on two levels. Firstly, on the

fUML model level, where fUML optimization rules (“fUML-Opti-Rules”) are applied to provide the modeller with advice to optimize his fUML model. Secondly, on the CSP model level, optimization rules (“CSP-Opti-Rules”) are applied to generate another more optimized CSP model. In this paper we propose a framework that integrates optimization with the formalization of fUML to CSP. The fUML-Opti-Rules are applied automatically by an Optimization Advisor component which generates some directions (advice) to the modeller that guides him to do the optimization manually, while a Model Optimizer component applies the CSP-Opti-Rules to generate an optimized CSP model. The paper does not introduce a new optimization algorithm that can be applied on the state space level because we do not have internal access to FDR2.

Our optimization is not applied on arbitrary CSP models (not generic optimization rules), rather it is applied on a generated CSP model that follows certain formalization rules (defined in [3]) and built of a subset of the CSP language. The main contribution of this work is that we seize the opportunity of having such a constrained CSP model to develop optimization rules that lead to a very reduced state space. Although the optimization rules do not preserve the original CSP model semantics, they have been proved to preserve the deadlock freedom property (the focus of this paper). Such specialized rules allowed for boosting the optimization to new areas that are hard to reach with the generic ones.

We implement this framework as a MagicDraw¹ [6] plugin called “Compass” to allow modellers to seamlessly use our framework during the system modelling process. The framework is based on Epsilon [7] to do the MDE (Model Driven Engineering) tasks, such as the model validation and the model-to-model transformation, supported by the fUML and CSP meta-models available in [1] and [8] respectively.

In order to realize and validate our approach, we modelled the GSS (Gas Station System) [9] and the CCS (Cruise Control System) [10] case studies in fUML, and then we used Compass to formalize, optimize and then check the model using FDR2. The GSS fUML model consists of nine objects communicating with each other asynchronously, while the CCS consists of seven objects. The behaviour of each object is modelled as an fUML activity diagram, parts of three of them are included in this paper. We do not consider the class diagram or any of its relations as our focus is on the behavioural analysis rather than the structure of the system. The results² of applying the optimization rules are outlined in this paper, and at the end of Section 6.3 we summarize the optimization results.

The rest of this paper is organized as follows. In Section 2, we give a brief introduction about fUML and CSP. In Section 3, we give an overview about the framework that applies our optimization approach. In Section 4, Section 5 and

¹ MagicDraw is an (award-winning) architecture, software and system modeling case tool. It also supports additional plugins to increase its functionalities.

² All the model checking in this paper has been done on an Intel Core 2 Duo machine with 2 GB memory.

Section 6 we describe the Optimization Advisor, Model Formalizer and Model Optimizer components respectively of this framework. Finally, we discuss related work and conclude in Sections 7 and 8 respectively.

2 Background

2.1 fUML

As defined by the OMG, fUML [1] is a standard that acts as an intermediary between “surface subsets” of UML models and platform executable languages. The fUML subset has been defined to allow the building of executable models. Code-generators can then be used to automatically generate executable code (e.g., in Java) from the models. Another option is to use model-interpreters that rely on a virtual machine to directly read and run the model (e.g., Cameo Simulation Toolkit [10]).

The fUML standard includes class and activity diagrams to describe a system’s structure and behaviour respectively. Some modifications have been applied to the original class and activity diagrams in the UML2 specification [11] to meet the computational completeness of fUML. The modifications have been done by merging/excluding some packages in UML2, as well as adding new constraints. We list in [3] some examples of the differences between the UML2 and fUML activity diagrams.

The Inter-object Communication Mechanism in fUML

The inter-object communication in fUML is defined by clause 8 in the standard [1]. Such communication is conducted between active objects only. Active objects in fUML communicate asynchronously via signals (kind of classifier). This is achieved by associating an object activation with each object that handles the dispatching of asynchronous communications received by its active object. Figure 1 shows the structure related to the object activation.

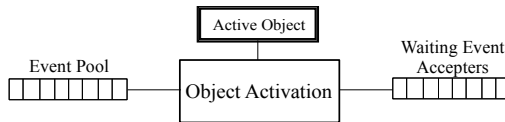


Fig. 1. Object Activation Structure

The object activation maintains two main lists: the first list (called *event pool*) holds the incoming signal instances waiting to be dispatched, and the second list (called *waiting event acceptors*) holds the event acceptors that have been registered by the executing classifier behaviour. Event acceptors are allowable signals with respect to the current state of the active object.

2.2 CSP

CSP [2] is a modelling language that allows the description of systems of interacting processes using a few language primitives. Processes execute and interact by means of performing events drawn from a universal set Σ . Some events are of the form $c.v$, where c represents a channel and v represents a value being passed along that channel. Our fUML formalization and the optimization theorem proofs consider the following subset of the CSP syntax:

$$\begin{aligned}
 P ::= & a \rightarrow P \mid c?x \rightarrow P(x) \mid d!v \rightarrow P \mid P_1 \square P_2 \\
 & \mid P_1 \sqcap P_2 \mid P_1 \parallel_A P_2 \mid P_1 \parallel P_2 \mid P \setminus A \\
 & \mid \text{let } N_1 = P_1, \dots, N_n = P_n \text{ within } P
 \end{aligned}$$

The CSP process $a \rightarrow P$ initially allows event a to occur and then behaves subsequently as P . The input process $c?x \rightarrow P(x)$ will accept a value x along channel c and then behaves subsequently as $P(x)$. The output process $d!v \rightarrow P$ will output v along channel d and then behaves as P . Channels can have any number of message fields, combination of input and output values.

The choice $P_1 \square P_2$ offers an external choice between processes P_1 and P_2 whereby the choice is made by the environment. Conversely, $P_1 \sqcap P_2$ offers an internal choice between the two processes.

The parallel combination $P_1 \parallel_A P_2$ executes P_1 and P_2 in parallel. P_1 and P_2 must simultaneously engage in events in the set A . $P_1 \parallel P_2$ is equivalent to $P_1 \parallel_{\alpha(P_1) \alpha(P_2)} P_2$, where P_1 and P_2 synchronize on the intersection between their alphabets' sets.

The hiding operation $P \setminus A$ describes the case where all participants of all events in the set A are described in P . All these events are removed from the interface of the process, since no other processes are required to engage in them. The *let ... within* statement defines P with local definitions $N_i = P_i$.

3 The Approach Framework Overview

Our optimization approach has been integrated with our previous formalization framework [4] by adding two extra components (the Optimization Advisor and the Model Optimizer), and separating the model-to-text task from the Model Formalizer to another component (CSP Script Generator). Figure 2 shows the comprehensive framework that performs the optimization and the formalization tasks, however we will focus in this paper on the optimization components only.

Initially, the modeller uses a case tool (e.g., MagicDraw) to develop the fUML model of the system and then chooses the property that he wants to check (deadlock freedom). The Optimization Advisor reads the fUML model and searches for specific patterns that are inefficient in terms of the resultant state space of the CSP model. The advice is reported to the modeller through an Optimization Report, so he can modify the fUML model and start the model formalization.

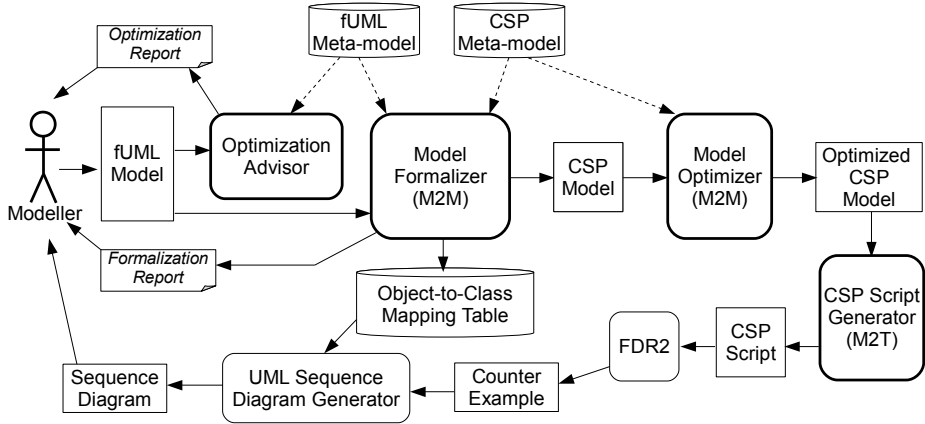


Fig. 2. The Integration of the Optimization Components

The Model Formalizer does the model-to-model transformation from fUML to CSP automatically based on their meta-models and a group of formalization rules to generate a CSP model (not script) that represents the fUML model behaviours. It also generates the Object-to-Class mapping table which is used for traceability by another component. At this stage, the CSP model is represented in an XML file format. Based on the chosen property, the Model Optimizer starts its function by reading the CSP model and applying a group of optimization rules automatically. Those rules transform the initial CSP model to an optimized one that still contains the required information to check the chosen property.

All the optimization rules included in this paper are independent (i.e., no rule depends on another one) and the more rules the modeller applies the more reduction in the state space he should get.

The CSP Script Generator component performs the model-to-text task that generates a CSP script from the input optimized CSP model automatically based on the same CSP meta-model. At this point FDR2 can be launched to do the model checking and if the checked property (deadlock) is not met, FDR2 generates a counter-example. The UML Sequence Diagram Generator reads the counter-example and uses the Object-to-Class mapping table to generate a UML sequence diagram that represents the counter-example in a modeller friendly format.

4 The Optimization Advisor

The fUML model may contain some patterns that are appropriate from the modeller's point of view and the system specification. However, when model checking the CSP representation of this fUML model, a state space explosion problem may happen. The focus here is on the patterns that cannot be removed automatically because certain decisions will be required from the modeller to avoid them.

The Optimization Advisor component scans the fUML model with regards to those patterns, and if found, it reports advice to the modeller to avoid them. We use EVL (Epsilon Validation Language) [7] together with the fUML meta-model to perform this task. The EVL script consists of a group of constraints, each one examines the existence of a certain pattern within a specific context. The following sub-sections describe two fUML-Opti-Rules and their optimization effect.

4.1 fUML-Opti-Rule(1): Detecting Self-sending Signals

When an object sends a signal to itself, we call this “self-sending”. Although this sounds benign, we found by experiment that self-sending signals cause an extra overhead on the object’s *event pool* buffer. Generally, a self-sending signal can be replaced by a direct control flow edge that joins the points of sending and accepting this signal. This replacement is safe provided that there are no actions between the two points (as they will be bypassed).

Another case is when the *AcceptEventAction* accepts the self-sending signal (beside another signal), such as the one highlighted in Figure 3 for the Pump object activity, because a direct control flow between the *FuelLevelLow* signal sending and acceptance (shown as a dashed line) does not preserve the behavioural semantics of the original activity unless the self-sending signal has the higher priority in the *event pool* (i.e., the first one to be dispatched from the object’s *event pool*). For that reason, we do not apply this fUML-Opti-Rule automatically. The Optimization Advisor just highlights the self-sending signals through the Optimization Report, leaving the choice to the modeller to do the removal based on his understanding of the fUML model.

On the other hand, it is obvious that removing the *CustomerFinished* signal sending and acceptance and replacing them with a direct control flow (shown as a dashed line at bottom of the diagram) will not affect the overall behaviour of the object.

We have developed an EVL constraint to check this pattern and report the advice to the modeller. The following EVL constraint applies fUML-Opti-Rule(1) on any *SendSignalAction*. The *if* condition checks if the target object of this send action is the sender object. The *message* field defines what the modeller will see in the Optimization Report in case that this constraint was not met.

```
context ActivityDiagram!SendSignalAction {
  constraint fUML-Opti-Rule-1{
    check {
      if(self.target.incoming.source.type.name = self.owner.name)
      {return false;}
      else{ return true; } }
    message : "The signal action '" + self.name + "' sends the signal to
      its object which can be replaced by a direct control flow." }}
```

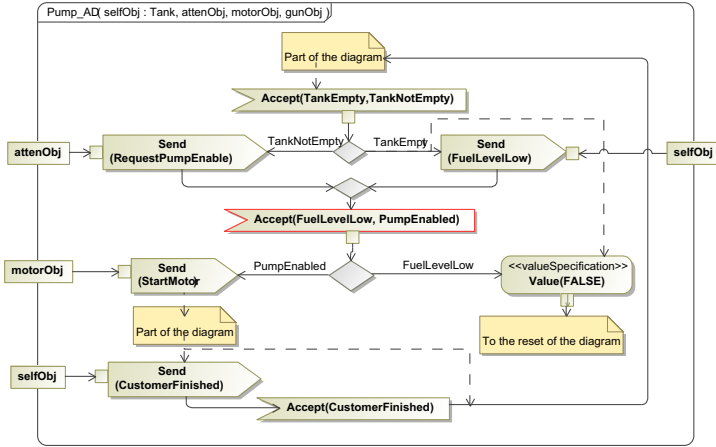


Fig. 3. Part of the Pump object activity

The Optimization Advisor managed to detect 4 self-sending signals in the GSS fUML model, and by safely replacing them with direct control flows the state space was reduced from 10.2M states to 4.7M states when checking the CSP formal representation of that fUML model.

4.2 fUML-Opti-Rule(2): Detecting Unacknowledged Signals

An “unacknowledged” signal is one that has been sent from an object to another object, and then it (source object) continues sending further signals without waiting for an acknowledgment signal. The problem arises when this pattern is repeated several times, because the system will be flooded with the unacknowledged signals and thus the objects’ *event pool* buffers will overflow.

As an example in the GSS, the Meter object activity was doing nothing but sending the *FuelUnitDelivered* signal to the Delivery object causing its *event pool* to overflow. If we acknowledged this signal by adding an accept event action for the *FuelUnitDeliveredACK* signal after the sending action, the Meter object will be forced to wait until the Delivery object sends the acknowledgment. Acknowledging signals does not convert the system from asynchronous to synchronous one, but it just controls the communicated signals in order to reduce the state space.

The Optimization Advisor uses the fUML-Opti-Rule(2) (represented in EVL) to scan the fUML model searching for the unacknowledged signals and report their existence through the Optimization Report. It is the modeller responsibility to acknowledge the signals in the fUML model.

fUML-Opti-Rule(2) helped in detecting three unacknowledged signals in the GSS fUML model. Acknowledging those signals reduced the state space of the

corresponding CSP model from 4.7M to 3.0M states when checking deadlock freedom.

5 The Model Formalizer

The Model Formalizer’s main function is to automatically transform the fUML model to a CSP formal representation that captures its behaviour. The transformation is done using ETL (Epsilon Transformation Language) which requires the source fUML meta-model and the target CSP meta-model available in [1] and [8] respectively. In our previous work [3] we presented a group of formalization rules which maps between the fUML activity diagram elements and their corresponding CSP. In [4] we described how we used a model-to-model transformation technique to apply those formalization rules automatically using ETL.

Our focus in this paper is on the optimization rather than the formalization, so we will not discuss the formalization rules and limit our discussion to the final result (CSP model) when applying the formalization rules on an fUML activity diagram. As an example for this application, consider the Tank object activity diagram in Figure 4. The diagram shows part of the activity which initially waits for the *FuelUsed* or the *RequestTankStatus* signals. If the Tank object accepted the *FuelUsed* signal, it subtracts the received amount from the current *tankLevel* by executing the *ReduceLevel* CallBehaviour action, then it checks the current level to set the *tankEmptyFlag* to *TRUE* if the level is below a certain threshold. If the Tank object accepted the *RequestTankStatus* signal (as a query from the Pump object), the *CheckLevel* CallBehaviour action is executed so the object sends a *TankEmpty* signal in the case where the action returned *TRUE*, otherwise it sends a *TankNotEmpty* signal.

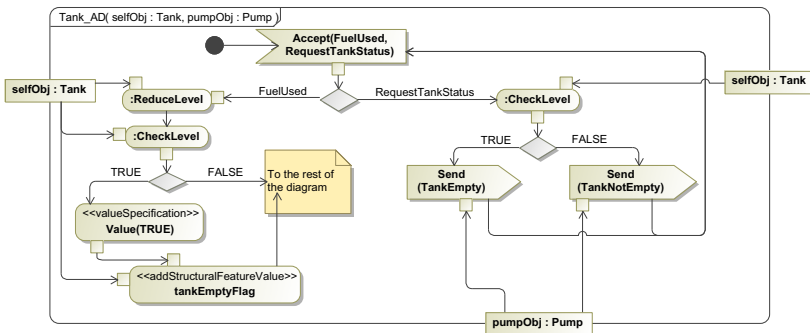


Fig. 4. Part of the Tank object activity

The Model Formalizer generates a CSP model by applying the ETL formalization rules on the Tank fUML activity diagram. This CSP model can be represented as the following CSP localized process that captures the Tank activity behaviour:

```

Tank_AD_Proc(selfObj, pumpObj) =
let
  AC1 = registerSignals!selfObj!rp1 → AC3
  AC3 = accept!selfObj!FuelUsed → AC2
      □
      accept!selfObj!RequestTankStatus → AC4)
  AC2 = AC5
  AC5 = AC6
  AC6 = TRUE!selfObj → AC7
      □
      FALSE!selfObj → ...
  AC7 = valueSpecification!selfObj?var : TRUE → AC8(var)
  AC8(var) = addStructuralFeatureValue!selfObj!tankEmptyFlag!var → ...
  AC4 = AC9
  AC9 = TRUE!selfObj → AC10
      □
      FALSE!selfObj → AC11
  AC10 = send!selfObj!pumpObj!TankEmpty → AC1
  AC11 = send!selfObj!pumpObj!TankNotEmpty → AC1
within AC1

```

The formalization of the fUML model includes also a formalization of the fUML inter-object communication mechanism (described in Section 2.1 and the formalization in [3]) that manages the signals sending, acceptance and dispatching through the *event pool* and the *waiting event accepters* list. The explanation of the localized CSP process below considers the formalization of this mechanism.

Initially, the *registerSignals* event adds the two signals (*FuelUsed* and *RequestTankStatus*) to the *waiting event accepters* list of the Tank object using the registration point *rp1*. In *AC3*, the *accept* event is not enabled until one of the two signals arrives to the object's *event pool*. The internal behaviour of the *ReduceLevel* and *CheckLevel* CallBehaviours will not affect the overall behaviour of the object, for that reason the Model Formalizer abstracts them into the processes *AC2*, *AC5* and *AC4* (which just enabling the next sub-process), and converts the decision based on their outputs to an internal choice (in *AC6* and *AC9*) so the model checker explores all the possible outputs. *AC7* and *AC8* are direct formalization for the *ValueSpecificationAction* and the *AddStructuralFeatureAction* respectively. Finally, the *send* event synchronizes with the other objects to insert signals in their *event pools* (e.g., *TankEmpty* in the Pump object's *event pool*).

6 The Model Optimizer

After the Model Formalizer finishes its function, the Model Optimizer starts the automatic optimization of the generated CSP model. The Model Optimizer performs a further transformation to the CSP model so that the model checker (FDR2) requires less states and time to check it. This transformation is done by applying a group of optimization transformation rules (CSP-Opti-Rules).

The CSP-Opti-Rules are defined in ETL also to perform such model-to-model transformation task based on the same CSP meta-model. Because the CSP-Opti-Rules are applied automatically, we support each one with a mathematical proof (theorem) to prove it is sound in that it does not eliminate important information that is required for checking a specific property (deadlock).

As mentioned in the introduction, our CSP-Opti-Rules are not generic (i.e., they cannot be applied on all CSP models). We make use of the opportunity of having a constrained CSP model that has been generated from specific rules that consider only a subset of the CSP language. Also, our CSP-Opti-Rules are constrained with the checked property (deadlock freedom). The following sub-sections show our three CSP-Opti-Rules and their effect on the state space.

6.1 CSP-Opti-Rule(1): Removing Passive Processes

We differentiate between two types of objects. First, core objects, which include the main behaviour of the system and interact with other objects in both directions (sending and accepting signals). Second, terminal objects, which represent external entities; however they interact with the system. For example, the GSS includes an object for the Attendant to simulate his interaction with the Pump; however, it is a terminal object because it will not be part of the system implementation. To check the model against deadlock freedom, the modeller should include all kinds of objects (core and terminal) in the fUML model to be able to explore all the system behaviours.

Passive objects are special kinds of the terminal objects as they interact in one direction (accepting signals only). The Motor object is one obvious example of the passive objects in the GSS model as it does nothing but accept signals (*StartMotor* and *StopMotor* signals) from the Pump.

In the CSP domain, a passive process is defined as the process that represents the passive object behaviour, which is always willing to interact (never refuses any interaction). On the implementation level, CSP-Opti-Rule(1) is represented as an ETL rule that scans the CSP model for any passive process, and if found, removes it from the CSP model. It removes the passive process from the parallel combination between the system's process which forms the SYSTEM big process. CSP-Opti-Rule(1) rules out a process to be passive if it contains the *send* event, which moves out the process from the passive condition (accepts signals only).

To formally verify that the removal of passive processes will not affect the deadlock checking of the system, we proved the following theorem for the passive process P_2 :

Theorem 1. *If P_2 is non-divergent and $(s, X_{P_2}) \in \mathcal{F}(P_2) \Rightarrow X_{P_2} = \{\}$ (passive process), then for any process P_1 : P_1 is deadlock free $\Leftrightarrow P_1 \parallel P_2$ is deadlock free.*

The assumption that P_2 is non-divergent is guaranteed by the definition of the formalization rules. Refer to [12] for the detailed proof of this theorem.

In the GSS fUML model, there is one passive object (the Motor object) and thus one passive process. When checking deadlock freedom, without applying

CSP-Opti-Rule(1) (or any other optimization rule) on the corresponding CSP model, FDR2 was unable to complete the check without crashing. After applying CSP-Opti-Rule(1), the Model Optimizer removed the Motor process the CSP model and FDR2 succeeded to report the deadlock freedom after exploring 12.3M states (the whole model). The reason of this reduction is the removal of the Motor object’s buffer which was synchronizing with the *send* events in the Pump object and hence the *send* event is considered as an external communication (i.e., turning a closed system into an open one).

6.2 CSP-Opti-Rule(2): Removing Abandoned Events

The global target of this rule is to search the CSP model for any event that can be removed from the model without affecting the deadlock checking results. We identify one kind of event that meets this criteria which we call “abandoned events”. Abandoned events are those which do not synchronize with any other events in another processes in the system. The removal is done by skipping to the next event/process as shown in the example below:

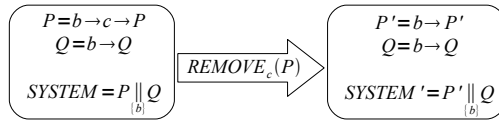


Fig. 5. The effect of CSP-Opti-Rule(2) on P

Theorem 2 is the formal representation of CSP-Opti-Rule(2). The theorem shows the constraints that make c an abandoned event, which is not to be member in any other process alphabets (i.e. no process will synchronize on c). It also shows that the removing of c from P_1 ($REMOVE_c(P_1)$) will not affect the deadlock checking result.

Theorem 2. *If $c \in \alpha(P_1)$, $c \notin \alpha(P_2)$, $P_1 \setminus c$ is non-divergent and $REMOVE_c(P_1)$ is defined then:*

$REMOVE_c(P_1) \parallel P_2$ is deadlock free $\Leftrightarrow P_1 \parallel P_2$ is deadlock free.

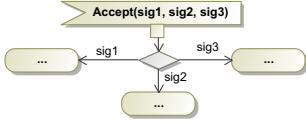
The proof of Theorem 2 (available in [12]) applies when P_1 is generated from the following grammar: $P ::= c \rightarrow P \mid c?x!y \rightarrow P(x) \mid P \square Q \mid P \sqcap Q$.

Additionally, in the case of $P \square Q$, the condition $c \notin initials(P) \wedge c \notin initials(Q)$ should be met, which is guaranteed because we do not use external choice except in the formalization rule (Rule(4) in [3]) in Table 1.

In other words, it is safe to apply CSP-Opti-Rule(2) on any generated CSP model from the Model Formalizer and by the correct selection of the abandoned event (e.g., not to be the *accept* event), provided that the deadlock freedom is the checked property.

We have implemented CSP-Opti-Rule(2) in ETL and applied it on the *value-Specification* and *addStructuralFeatureValue* events, because they are not synchronizing with any other events and they never happen after an external choice.

Table 1. fUML to CSP mapping rule

| fUML Element | CSP Representation |
|--|---|
| <p><i>Rule(4): Accept Event Action (*)</i></p>  | <pre> AC1 = registerSignals!bIH!rp2 → (accept!bIH!sig1 → ... □ accept!bIH!sig2 → ... □ accept!bIH!sig3 → ...) </pre> |

The elimination of the abandoned events reduces the state space size, and thus allows faster FDR2 checks. When CSP-Opti-Rule(2) was applied on the GSS corresponding CSP model, 8 abandoned events were removed to reduce the state space from 12.3M to 10.2M states. Using the standard CSP hiding operator instead of $REMOVE_c(P)$ did not provide such reduction in the state space because the hiding does not remove the event, it just renames it to τ .

6.3 CSP-Opti-Rule(3): Toggling Internal Choices

Unlike the first two CSP-Opti-Rules, this one needs human input before its automatic application. It also does not lead to an optimized version of the original model. Rather, it splits the original model into sub-models that are easier to be checked separately using FDR2. This kind of CSP-Opti-Rules is very useful when analyzing big models, as it will allow the modeller to focus on certain parts of the model at a time. This is a bounded approach to find and solve the model’s problems. Another benefit is that when the model is too big to be analyzed by FDR2, the CSP-Opti-Rule can be used to analyze the system in different stages, each stage is an analysis of one of the sub-models.

CSP-Opti-Rule(3) can be summarized as follows: when checking deadlock, if all the sub-models are deadlock free, then the original model is deadlock free as well. The splitting up of the original model is done based, generally, on reducing the behavioural paths in the model’s processes. In particular, CSP-Opti-Rule(3) replaces an internal choice with one direct connection to one of its choices. And in the case of more than two choices, it disables one of them. The selection of the enabled choice(s) comes from the modeller input through a Graphical User Interface (GUI).

To apply this rule, the Optimization Advisor scans the fUML model for the decision nodes that will be translated to internal choices in the CSP model and builds a table with those nodes/choices. After building the table, the modeller can use the GUI to toggle the choice branches and start the model checking. The selected choices will be passed to the Model Optimizer that applies CSP-Opti-Rule(3) based on the modeller selection. After the model checking, the modeller can repeat the process with different choice(s). Illustrated below Theorem 3,

which proves that this accumulative deadlock checking (on several stages) is equivalent to the original model deadlock checking (as a whole).

Theorem 3. *If $SYS = P_1 \sqcap P_2$, $SYS' = P_1$ (after splitting SYS and choosing the first branch), and $SYS'' = P_2$ (after splitting SYS and choosing the second branch), then:*

$$SYS \text{ Deadlock Free} \Leftrightarrow SYS' \text{ Deadlock Free} \wedge SYS'' \text{ Deadlock Free}$$

As a sample usage of this rule in the GSS case study, assume that the modeller is doing some modifications in the fUML model and he wants to repeat the model checking several times to ensure that the modifications do not affect the deadlock freedom of the system. Using CSP-Opti-Rule(3) to disable the tank emptiness choice (i.e., the tank can never be empty), FDR2 managed to check the model for deadlock in 11 minutes after exploring 1.8M states instead of 18.4 minutes and 3.0M states when the two choices are available (i.e., the tank can be empty or not). Finally, the modeller should enable the other branch of the same choice (i.e., the tank is always empty) to ensure the system deadlock freedom (done in 15 minutes and 2.6M states).

The following table summarizes the results of applying our approach on the GSS and the CCS case studies using Compass. The “States” field shows the explored number of states by FDR2 until reporting the deadlock freedom of the model. The order of the applied rules is just according to our test scheme. It is obvious that applying the CSP-Opti-Rules and the advice from the fUML-Opti-Rules led to a substantial reduction in the state space and the model checking time.

Table 2. Optimization results for the GSS and CCS case studies

| fUML-Opti-Rule(1) | fUML-Opti-Rule(2) | CSP-Opti-Rule(1) | CSP-Opti-Rule(2) | CSP-Opti-Rule(3) | GSS | | | | CCS | | | |
|-------------------|-------------------|------------------|------------------|------------------|--------|-------|----------|--------|--------|-------|----------|----------|
| | | | | | States | | Time | | States | | Time | |
| | | ✓ | | | 12.3 M | | 1.38 H | | 38.2 M | | 2.7 H | |
| | | ✓ | ✓ | | 10.2 M | | 1.15 H | | 36.0 M | | 2.5 H | |
| ✓ | | ✓ | ✓ | | 4.7 M | | 29.7 Min | | 25.1 M | | 1.7 H | |
| ✓ | ✓ | ✓ | ✓ | | 3.0 M | | 18.4 Min | | 7.1 M | | 28.5 Min | |
| ✓ | ✓ | ✓ | ✓ | ✓ | 1.8 M | 2.6 M | 11.0 Min | 15 Min | 5.2 M | 6.4 M | 20.6 Min | 25.5 Min |

7 Related Work

There is a significant body of work researched in avoiding the model checking state explosion problem. Some authors such as Planas *et al.* [13] avoided the problem completely by using static analysis which cannot be used for analysing the dynamic behaviour of the system’s object. The others who preferred the model checking, can be categorized as follows:

The first category includes the work that focuses on optimizing the corresponding LTS (Label Transition System) of the formal model. FDR2 (and many other model checkers), works by calculating the LTS semantics of the CSP processes and then perform the model checking on the LTS level [14]. The bisimulation minimization in FDR2 [15] is an optimization technique that lies in this category. The supercompilation [16] is another example where FDR2 calculates a set of rules for turning a combination of LTS's into a single LTS, without explicitly constructing it. Also, Roscoe in [14] showed how to use FDR2 compression functions such as: *sbisim*, *normal* and *diamond* to compress the LTS of the CSP model. ProB [17] is another model checker that uses another optimization techniques on the LTS level such as permutation flooding [18] and hash value [19] symmetric reduction.

The second category includes those who concentrate on the formal model optimization before translating it to the LTS representation. Decomposing the formal model into constituent parts to have an effective model checking is one of the techniques that has been used in this category. One such example is Wang *et al.* [20] who proposed using Extended Hierarchical Automata (EHA) for UML state diagrams formalization, and then slice the EHA model based on a slicing criterion extracted from the checked property. Another example is Schneider *et al.* [21] who proposed decomposing the CSP||B model into finer grained components called *chunks* to allow checking divergence freedom in large systems using FDR2. Apart from the decomposition, some data abstraction techniques can also be used to optimize the formal model. For example, Jesus *et al.* [22] abstract any infinite domain in the system to allow checking the CSP models using FDR2.

Our work lies in the second category, especially when considering the Model Optimizer component which applies the CSP-Opti-Rules directly. However, we could not find in the literature an approach that provides optimization advice on the semi-formal model level before the formalization. Also implementing a comprehensive framework to apply the formalization and optimization based on an MDE technique is a distinguished point for our work.

8 Conclusion

We have described in this paper a framework for optimizing the formal representation (CSP) of the fUML models. The framework does the formalization and the optimization tasks using different components. We described two of those components that applies the optimization rules. The first one is the Optimization Advisor which uses EVL to provide the modeller with some advice to avoid some undesirable patterns in his fUML model. The second component is the Model Optimizer which uses ETL to generate an optimized CSP model. The results of applying the specialized optimization rules showed a substantial reduction in the state space, and thus in the model checking time. We would argue that the general idea of our optimization approach is applicable for the other work which consider the UML formalization.

Currently, the included five optimization rules are all that we have developed and verified; however, more optimization rules that address other generic (e.g.,

livelock freedom or states reachability) and non-generic properties will be considered in future work. Also we will conduct further verification of the optimization approach via more case studies that challenges its capabilities.

References

1. OMG: Semantics of a foundational subset for executable UML models (fUML) - Version 1.0 (February 2011)
2. Schneider, S.: *Concurrent and Real-Time Systems: the CSP Approach*. Wiley (1999)
3. Abdelhalim, I., Sharp, J., Schneider, S.A., Treharne, H.: Formal Verification of Tokeneer Behaviours Modelled in fUML Using CSP. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 371–387. Springer, Heidelberg (2010)
4. Abdelhalim, I., Schneider, S.A., Treharne, H.: Towards a Practical Approach to Check UML/fUML Models Consistency Using CSP. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 33–48. Springer, Heidelberg (2011)
5. *Formal Systems Oxford: FDR 2.91 manual* (2010)
6. MagicDraw CASE tool, <http://www.magicdraw.com/>
7. Kolovos, D., Rose, L., Paige, R.: *The Epsilon Book*
8. Treharne, H., Turner, E., Paige, R.F., Kolovos, D.S.: Automatic Generation of Integrated Formal Models Corresponding to UML System Models. In: *TOOLS*, vol. (47), pp. 357–367 (2009)
9. Raistrick, C., Francis, P., Wright, J., Carter, C., Wilkie, I.: *Model Driven Architecture with Executable UML*. Cambridge University Press (2004)
10. Cameo Simulation Toolkit, <https://www.magicdraw.com/simulation>
11. OMG: Unified modeling language (UML) superstructure, version 2.3 (2010)
12. Abdelhalim, I., Schneider, S.A., Treharne, H.: Formalized fUML Models Optimization. Technical Report, University of Surrey, Department of Computing Technical Report CS-12-04 (June 2012)
13. Planas, E., Cabot, J., Gómez, C.: Verifying Action Semantics Specifications in UML Behavioral Models. In: van Eck, P., Gordijn, J., Wieringa, R. (eds.) *CAiSE 2009*. LNCS, vol. 5565, pp. 125–140. Springer, Heidelberg (2009)
14. Roscoe, A.W.: *Understanding Concurrent Systems*. Springer (2010)
15. Roscoe, A.W.: *The Theory and Practice of Concurrency*. Prentice Hall (1998)
16. Goldsmith, M.: *Operational Semantics for Fun and Profit*, pp. 265–274 (2005)
17. The ProB Animator and Model Checker, <http://www.stups.uni-duesseldorf.de/ProB>
18. Leuschel, M., Butler, M., Spermann, C., Turner, E.: Symmetry reduction for B by permutation flooding. In: Julliand, J., Kouchnarenko, O. (eds.) *Optimization Techniques 1973*. LNCS, vol. 4, Springer (January 2007) copyright Springer
19. Leuschel, M., Massart, T.: Efficient approximate verification of B via symmetry markers. In: *The International Symmetry Conference* (2007)
20. Wang, J., Dong, W., Qi, Z.-C.: Slicing Hierarchical Automata for Model Checking UML Statecharts. In: George, C.W., Miao, H. (eds.) *ICFEM 2002*. LNCS, vol. 2495, pp. 435–446. Springer, Heidelberg (2002)
21. Schneider, S., Treharne, H., Evans, N.: Chunks: Component Verification in CSP||B. In: Romijn, J.M.T., Smith, G.P., van de Pol, J. (eds.) *IFM 2005*. LNCS, vol. 3771, pp. 89–108. Springer, Heidelberg (2005)
22. Jesus, J., Mota, A., Sampaio, A., Grijo, L.: Architectural verification of control systems using CSP. In: Qin, S., Qiu, Z. (eds.) *ICFEM 2011*. LNCS, vol. 6991, pp. 323–339. Springer, Heidelberg (2011)

Efficient Probabilistic Abstraction for SysML Activity Diagrams

Samir Ouchani*, Otmane Ait Mohamed, and Mourad Debbabi

Computer Security Laboratory, Hardware Verification Group,
Concordia University, Montreal, Canada
{s_oucha,ait,debbabi}@ece.concordia.ca

Abstract. SysML activity diagrams are OMG/INCOSE standard models for specifying and analyzing systems' behaviors. In this paper, we propose an abstraction approach for this type of diagrams that helps to mitigate the state-explosion problem in probabilistic model checking. To this end, we present two algorithms to reduce the size of a given SysML activity diagram. The first eliminates the irrelevant behaviors regarding the property under check, while the second merges control nodes into equivalent ones. The resulting abstracted model can answer safely the Probabilistic Computation Tree Logic (PCTL) property. Moreover, we present a novel calculus for activity diagrams (NuAC) that captures their underlying semantics. In addition, we prove the soundness of our approach by defining a probabilistic weak simulation relation between the semantics of the abstract and the concrete models. This relation is shown to preserve the satisfaction of the PCTL properties. Finally, we demonstrate the effectiveness of our approach on an online shopping system case study.

Keywords: Abstraction, SysML Activity Diagram, Probabilistic Automata, PCTL.

1 Introduction

Various techniques have been proposed for the verification of software and systems including model checking, type checking, equivalence checking, theorem proving, and dynamic analysis. Particularly, the most popular one used for the assessment of UML and SysML behavioral diagrams is model checking [12]. The latter is a formal automatic verification technique for finite state concurrent systems that checks temporal logic specifications on a given model. In addition to qualitative model checking, quantitative verification techniques based on probabilistic model checkers [3] have recently gained popularity. Probabilistic verification offers the capability of interpreting probabilistically the satisfiability of a given property on systems that inherently exhibit probabilistic behavior. Despite its wide use, model checking is generally a resource-intensive process that requires large amount of memory and time processing. This is due to the fact that the systems' state space may grow exponentially with the number of variables combined with the presence of concurrent behaviors and clocks. Moreover, several

* Corresponding author.

available model checkers cannot support all systems' features such as buffers, channels, and/or real variables. To overcome these issues, various techniques have been explored [4] for qualitative model checking and then leveraged to the probabilistic case. Among these techniques, several solutions aim at optimizing the employed model checking algorithms by introducing symbolic data structures based on binary decision diagrams, while others target the reduction of the input model. As we are interested in reusing existing model checkers, we concentrate on the second category that includes abstraction approaches.

Abstraction is one of the most relevant technique for addressing the state explosion problem [3,5]. It can be defined as a mapping from a concrete model into a more abstract one that encapsulates the systems' behavior while being of a reduced size. The intuition behind this transformation is to be able to check a property against an abstract model and then to infer safely the same result on the concrete model. Abstraction techniques can be classified in four categories [6]: 1) *Abstraction by state merging* aims at merging states of systems that have similar features. 2) *Abstraction on variables* targets the data in the model and aims at representing a set of values as one symbolic variable. 3) *Abstraction by restriction* operates by forbidding some behavior of the system. 4) *Abstraction by observer automata* restricts system's behaviors to those acceptable by an automaton that observes the system from outside. Our proposed framework takes advantage of the first and the third category.

In this paper, we are interested in the efficient verification of systems' design models expressed as SysML activity diagrams [7]. These diagrams are behavioral and allow for probabilistic behavior specification. Our approach combines two mechanisms of abstraction, the first is based on ignoring the irrelevant action nodes with respect to a given property and the second applies reduction rules that collapse control nodes in the SysML activity diagram. Our approach is depicted in Figure 1. In order to prove the soundness of our algorithm and the preservation of PCTL [3,8] properties satisfaction, we present a new calculus, namely NuAC, that captures the semantics foundation of SysML activity diagrams and we express its operational semantics as probabilistic automata. Furthermore, we demonstrate the practical application of our technique using a case study that would be otherwise difficult to verify. Thus, we use the probabilistic model checker PRISM [9] and we rely on our translation algorithm [2] that maps SysML activity diagrams into PRISM model. Besides, we show significant reduction in the state space and verification time, which makes probabilistic model checking helpful.

The remainder of this paper is organized as follows. Section 2 presents the related work. Section 3 explains the formal representation of SysML activity diagrams. The proposed abstraction approach is detailed in Section 4. Section 5 defines and proves the soundness of our algorithm and the preservation of PCTL satisfaction. Section 6 describes the experimental results. Finally, Section 7 concludes this paper and provides hints on the possible future works.

2 Related Work

In the literature, few works examine the abstraction of UML and SysML diagrams before verification and the majority rely on the implemented abstraction algorithms within the model checker.

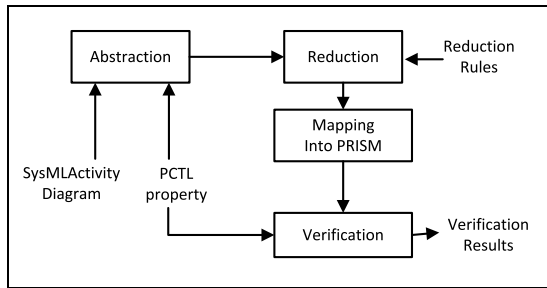


Fig. 1. A Novel Abstraction Approach Overview

Ober et al. [10,11] propose a set of UML model reduction techniques including static analysis, partial order reduction, and model minimization. The abstraction is performed on the semantic model instead of UML diagrams. Westphal [12] exploits the symmetry of UML models in the type of object references to do verification. Shet [13] determines the set of relevant events with respect to the safety property. Daoxi et al. [14] propose an abstraction driven by LTL properties on Promela code of UML behavioral diagrams. Xie and Browne [15,16] propose a verification framework for executable UML (xUML) models. It is based on a user-driven state space reduction procedure. Beek et al. present in [17] a framework called (UMC) for the formal analysis of concurrent systems specified by a collection of UML state machines. It is an on-the-fly based analysis with a user-guided abstraction of the transition system. Gallardo et al. [18] abstract data and events in hierarchical state chart diagrams. They minimize the original access definitions of variables and use a single event name to represent a set of real ones. R. Eshuis [19,20] apply data abstraction on guards and events. In addition, some probabilistic model checker support abstraction, for example PRISM builds the symmetry reduction and LiQuor¹ includes bi-simulation.

In Table 1, we compare our approach to the existing ones. We observe that few of them formalize SysML activity diagrams and prove the soundness of their proposed abstraction approaches. Moreover, our abstraction approach is efficient as it reduces the size of the model by a considerable rate. Furthermore, our mechanism allows to gain advantage from algorithms built within the tool in use.

3 SysML Activity Diagrams

In this section, we describe the SysML activity diagram notation, and we present an optimized and a modified version of the Activity Calculus (AC) [1], to provide a formal syntax and operational semantics for SysML activity diagrams. This formal semantics is useful to prove the soundness of our abstraction.

SysML reuses a subset of UML packages and extends others with specific systems' engineering features, and it covers four main perspectives of systems modeling: structure, behavior, requirements, and parametric. Particularly, SysML activity diagrams are

¹ <http://www.i1.informatik.uni-bonn.de/baier/projectpages/LIQUOR/LiQuor>

Table 1. Comparison with the Related Work

| Approach | Design | Probabilistic | Property | Formalization | Soundness |
|----------|--------|---------------|----------|---------------|-----------|
| [10][11] | | | | | |
| [12] | ✓ | | | | |
| [13] | ✓ | | ✓ | | |
| [14] | | | ✓ | | |
| [15][16] | ✓ | | | | |
| [17] | ✓ | | | | |
| [18] | ✓ | | | | |
| [19][20] | ✓ | | | | |
| Our | ✓ | ✓ | ✓ | ✓ | ✓ |

behavioral diagrams used to model system's behavior at various level of abstractions [21]. The main notation of SysML activity diagram can be decomposed into two categories of constructs: activity nodes and activity edges. The former contains three types: activity invocation, object and control nodes. Activity invocation includes receive and send signals, action, and call behavior. Activity control nodes are initial, flow final, activity final, decision, merge, fork, and join nodes. Activity edges are of two types: control flow and object flow. Control flow edges are used to show the execution path through the activity diagram and connects activity nodes. Object flow edges are used to show the flow of data between activity nodes. Concurrency and synchronization are modeled using forks and joins, whereas, branching is modeled using decision and merge nodes. While a decision node specifies a choice between different possible paths based on the evaluation of a guard condition (and/or a probability distribution), a fork node indicates the beginning of multiple parallel control threads. Moreover, a merge node specifies a point from where different incoming control paths follow the same path, whereas a join node allows multiple parallel control threads to synchronize and rejoin.

3.1 Syntax of SysML Activity Diagrams

The UML superstructure specifies basic rules for the execution of the various nodes by explaining textually how tokens (i.e. *locus of control*.) are passed from one node to another [22]. At the beginning, a first token starts flowing from the initial node and moves downstream from one node to another with respect to the foregoing set of control routing rules defined by the control nodes until reaching either an activity final or a flow final node. However, activity diagram semantics as specified in the standard stay informal since it is described informally using textual explanations. Inspired by this concept, we express the Backus-Naur-Form (BNF) of the new version of Activity Calculus (NuAC) that captures the syntax and the execution of activity diagrams. This new version optimizes the syntax presented in [1] and allows multiplicity for fork and decision constructs. Before presenting the NuAC syntax, firstly, we rewrite the SysML activity diagram constructs in the formal way as described in Table 2. The BNF of NuAC is illustrated in Figure 2.

During the execution, the structure of the activity diagram is kept unmodified but the location of the tokens changes. The NuAC syntax was inspired by this idea so that an

Table 2. Mapping Activity Diagram Artifacts into NuAC Syntax

| AD Constructs | NuAC Syntax | Description |
|---------------|--|---|
| | $t \mapsto \mathcal{N}$ | Initial node |
| | $l : \odot$ | Activity final node |
| | $l : \otimes$ | Flow final node |
| | $l : a \mapsto \mathcal{N}$ | Action node |
| | $l : \text{Decision}(\langle p_1, g_1, \mathcal{N}_1 \rangle, \dots, \langle p_n, g_n, \mathcal{N}_n \rangle)$ | a Decision node with a convex distribution $\{p_1, \dots, p_n\}$ with guarded transitions $\{g_1, \dots, g_n\}$ |
| | $l : \text{Merge}(\mathcal{N})$ or l | Merge node specifies the continuation. |
| | $l : \text{Fork}(\mathcal{N}_1, \dots, \mathcal{N}_n)$ | Fork node models the concurrency between n control threads. |
| | $l : x.\text{Join}(\mathcal{N})$ or l | Join node models synchronization. It rejoins a set of input pins. Each pin is specified by an index x . |

| | |
|---|---|
| $\mathcal{A} ::= \varepsilon$ | $\bar{t}^k \mapsto \mathcal{N}$ |
| $\mathcal{N} ::= \overline{\mathcal{N}}^n$ | $l : \text{Merge}(\mathcal{N}) \mid l : x.\text{Join}(\mathcal{N})$ |
| $\mid l : \text{Fork}(\mathcal{N}, \dots, \mathcal{N})$ | $\mid l : a \mapsto \mathcal{N}$ |
| $\mid l : \text{Decision}(\langle p_1, g_1, \mathcal{N} \rangle, \dots, \langle p_n, g_n, \mathcal{N} \rangle)$ | $\mid l : \odot$ |
| $\mid l : \otimes$ | $\mid l$ |

Fig. 2. Syntax of New Activity Calculus (NuAC)

NuAC term presents a static structure while tokens are the only dynamic elements. We can distinguish two main syntactic concepts: marked and unmarked terms. A marked NuAC term corresponds to an activity diagram with tokens. An unmarked NuAC term corresponds to the static structure of the diagram. A marked term is typically used to denote a reachable configuration. A configuration is characterized by the set of tokens locations in a given term.

To support multiple tokens, we augment the “overbar” operator with an integer n such that $\overline{\mathcal{N}}^n$ denotes a term marked with n tokens such that $\overline{\mathcal{N}}^1 = \overline{\mathcal{N}}$ and $\overline{\mathcal{N}}^0 = \mathcal{N}$. For the term \bar{t}^k , k can be either 1 or 0. Multiple tokens are needed when there are loops that encompass in their body a fork node. Furthermore, we use a prefix label for each node (except initial) to uniquely reference it in the case of a backward flow connection. Particularly, labels are useful for connecting multiple incoming flows towards merge and join nodes. Let \mathcal{L} be a collection of labels ranged over by l, l_0, l_1, \dots and \mathcal{N} be any node (except initial) in the activity diagram. We write $l : \mathcal{N}$ to denote an l -labeled

activity node \mathcal{N} . The NuAC term \mathcal{A} is built using a depth-first traversal of the activity diagram directed by its activity edges. It is important to note that nodes with multiple incoming edges (e.g. join and merge) are visited as many times as they have incoming edges. Thus, as a syntactic convention, the algorithm uses either the definition term (i.e. $l: Merge(\mathcal{N})$ for merge and $l: x.Join(\mathcal{N})$ for join) if the current node is visited for the first time or the corresponding label (i.e. l) if the same node is encountered later during the traversal process. We denote by $Decision(-, g, \mathcal{N})$ to express the non-probabilistic decision while p has no value. Also, we denote by $\mathcal{A}[\mathcal{N}]$ to specify \mathcal{N} as a sub term of \mathcal{A} .

3.2 Semantics of SysML Activity Diagrams

The execution of SysML activity diagrams is based on token's flow [22]. To give a meaning to this execution, we use structural operational semantics to formally describe how the computation steps of NuAC atomic terms take place. The NuAC semantics rules shown by Figure 3 is based on the informally specified tokens-passing rules defined in [22].

| | |
|-----------|--|
| INIT-1 | $\overline{T} \xrightarrow{1} \mathcal{N} \xrightarrow{l} \underline{T} \xrightarrow{1} \overline{\mathcal{N}}$ |
| ACT-1 | $\overline{l: a^m} \xrightarrow{1} \mathcal{N} \xrightarrow{l} \underline{T} \xrightarrow{1} \overline{l: a^{m-1}} \xrightarrow{1} \overline{\mathcal{N}} \quad \forall m > 0$ |
| ACT-2 | $\overline{l: a^m} \xrightarrow{1} \mathcal{N} \xrightarrow{l} \underline{T} \xrightarrow{1} \overline{l: a^{m+n}} \xrightarrow{1} \overline{\mathcal{N}} \quad \forall m > 0$ |
| FORK-1 | $\overline{l: Fork(\mathcal{N}_1, \dots, \mathcal{N}_n)^m} \xrightarrow{l} \underline{T} \xrightarrow{1} \overline{l: Fork(\overline{\mathcal{N}}_1, \dots, \overline{\mathcal{N}}_n)^{m-1}} \quad \forall m > 0$ |
| PDEC-1 | $\overline{l: Decision((p_1, g_1, \mathcal{N}_1), \dots, (p_i, g_i, \mathcal{N}_i), \dots, (p_n, g_n, \mathcal{N}_n))^m} \xrightarrow{l} \underline{T} \xrightarrow{p_i} \overline{l: Decision((p_1, g_1, \mathcal{N}_1), \dots, (p_i, g_i, \overline{\mathcal{N}}_i), \dots, (p_n, g_n, \mathcal{N}_n))^{m-1}} \quad \forall m > 0$ |
| MERG-1 | $\overline{l: Merge(\mathcal{N})^m} \xrightarrow{l} \underline{T} \xrightarrow{1} \overline{l: Merge(\overline{\mathcal{N}})^{m-1}} \quad \forall m > 0$ |
| MERG-2 | $\mathcal{A}[\overline{l: Merge(\mathcal{N})^m}, \underline{T}] \xrightarrow{l} \mathcal{A}[\overline{l: Merge(\mathcal{N})^{m+k}}, \underline{T}] \quad m, k \geq 1$ |
| JOIN-1 | $\mathcal{A}[\overline{l: x.Join(\mathcal{N})^m}, \underline{T}_x^k] \xrightarrow{l} \mathcal{A}[\overline{l: x.Join(\overline{\mathcal{N}})^{m+k-1}}, \underline{T}_x] \quad x > 1 \quad \forall m, k \geq 1$ |
| FLOWFINAL | $\mathcal{A}[\overline{l: \otimes}] \xrightarrow{l} \underline{T} \xrightarrow{1} \mathcal{A}$ |
| FINAL | $\mathcal{A}[\overline{l: \odot}] \xrightarrow{l} \underline{T} \xrightarrow{1} \mathcal{A}$ |
| PROG | $\mathcal{N} \xrightarrow{\alpha} \underline{q} \mathcal{N}'$ |
| | $\mathcal{A}[\mathcal{N}] \xrightarrow{\alpha} \underline{q} \mathcal{A}[\mathcal{N}']$ |

Fig. 3. NuAC Operational Semantic Rules

We define Σ as the set of non-empty actions labeling the transitions (i.e. the alphabet of NuAC, to be distinguished from action nodes in activity diagrams). An element $\alpha \in \Sigma$ is the label of the executing active node. Let Σ^o be $\Sigma \cup \{o\}$ where o denotes the empty action. Let p be probability values such that $p \in \{-\} \cup [0, 1]$. The general form of a transition is $\mathcal{A} \xrightarrow{\alpha} \underline{p} \mathcal{A}'$. The probability value specifies the likelihood of a given transition to occur and it is denoted by $P(\mathcal{A}, \alpha, \mathcal{A}')$. The semantics of SysML activity diagrams is expressed using \mathcal{A} as a result of the defined inference rules that can be described in terms of Probabilistic Automata (PA) [23].

4 The Abstraction Approach

This section describes our approach to abstract a design model expressed as SysML activity diagrams. To do so, we propose essentially two abstraction algorithms.

The first one hides the action nodes of the SysML activity diagram that are not part of the atomic propositions of the PCTL property to be verified à la [5]. Initially, SysML activity diagram is an action-based diagram, where actions are the executed entities and guards denote the branching choices between alternative actions. The control nodes are essentially used to coordinate the execution of these actions. Thus, PCTL properties essentially comprise propositions on actions and guards. The atomic propositions of a PCTL property ϕ are formed with a set of independent variables $var(\phi)$ such that $var(\phi) \subseteq \{a_i : i \leq n\} \cup \{g_i : i \leq m\}$ where a_i is a variable corresponding to an action, g_i is a guard variable, and respectively, n and m are the number of actions and guards.

```

Abs: NuAC × Var(Φ) → NuAC
Abs( $\mathcal{N}$ , var( $\phi$ )) = Case ( $\mathcal{N}$ ) of
     $l: a \mapsto \mathcal{N}' \quad \Rightarrow \quad$  if  $a \in var(\Phi)$  then
                                                 $l: a \mapsto Abs(\mathcal{N}', var(\phi))$ 
    else
                                                 $\varepsilon \mapsto Abs(\mathcal{N}', var(\phi))$ 
    end
    Otherwise  $Abs(\mathcal{N}, var(\phi))$ 
    
```

Listing 1.1. Action Nodes Abstraction Algorithm

The procedure *Abs* presented in Listing 1.1 abstracts a given SysML activity diagram with respect to the action variables that are not part of the set $var(\phi)$. It takes as input a NuAC term \mathcal{N} along with $var(\phi)$ and generates an abstract term such that $Abs(\mathcal{N}, var(\phi)) = \widehat{\mathcal{N}}$ and $var(\widehat{\mathcal{N}}) = var(\phi)$.

The second algorithm minimizes an activity diagram by merging specific control nodes while preserving the number of tokens and their control paths. This is achieved by preventing the modification of guarded and probabilistic choices. The procedure *Minim* presented in Listing 1.2 aims at merging consecutive control nodes of the same type. For that, we propose an equivalence relation inspired by the structural congruence relation defined by Milner [24]. This equivalence relation satisfies the following rules:

1. $l: Fork(\dots, \mathcal{N}_i, \dots) \equiv l: Fork(\dots, \mathcal{N}_k, \dots, \mathcal{N}_m, \dots)$ if $\mathcal{N}_i \equiv l': Fork(\mathcal{N}_k, \dots, \mathcal{N}_m)$.
2. $l: x.Join(\mathcal{N}') \equiv l: z.Join(\mathcal{N})$ if $\mathcal{N}' \equiv l': y.Join(\mathcal{N})$ and $z = x + y$.
3. $l: Merge(\mathcal{N}) \equiv l: Merge(\mathcal{N}')$ and $l = l'$ if $\mathcal{N} \equiv l': Merge(\mathcal{N}')$.
4. $l: Decision(\dots, (p, g, \mathcal{N}), \dots) \equiv l: Decision(\dots, (p \times p_k, g \wedge g_k, \mathcal{N}_k), \dots, (p \times p_m, g \wedge g_m, \mathcal{N}_m), \dots)$ if $\mathcal{N} \equiv l': Decision((p_k, g_k, \mathcal{N}_k), \dots, (p_m, g_m, \mathcal{N}_m))$.

Basically, *Abs* produces a new model that includes mainly the specified actions in the property ϕ where other actions are considered as silent action (Milner [24]). Thus, the

² Close to the cone of influence of [5].

```


$$\text{Minim: } NuAC \rightarrow NuAC$$


$$\text{Minim}(\mathcal{N}) = \text{Case } (\mathcal{N}) \text{ of}$$

  
$$l: \text{Merge}(\mathcal{N}') \Rightarrow \text{Case } (\mathcal{N}') \text{ of}$$

    
$$l': \text{Merge}(\mathcal{N}'') \Rightarrow l: \text{Merge}(\text{Minim}(\mathcal{N}'')) \text{ and } \text{Rewrite}(\mathcal{N}'', l', l)$$

    
$$\text{Otherwise} \Rightarrow l: \text{Merge}(\text{Minim}(\mathcal{N}'))$$

  
$$l: x.\text{Join}(\mathcal{N}') \Rightarrow \text{Case } (\mathcal{N}') \text{ of}$$

    
$$l': y.\text{Join}(\mathcal{N}'') \Rightarrow \text{let}$$

      
$$z = x+y$$

    
$$\text{in}$$

    
$$l: z.\text{Join}(\text{Minim}(\mathcal{N}'')) \text{ and } \text{Rewrite}(\mathcal{N}'', l', l_{x+j})$$

    
$$\text{Otherwise} \Rightarrow l: \text{Join}(\text{Minim}(\mathcal{N}'))$$

  
$$l: \text{Fork}(\mathcal{N}_1, \dots, \mathcal{N}_i, \dots, \mathcal{N}_n) \Rightarrow \text{Case } (\mathcal{N}_i) \text{ of}$$

    
$$l': \text{Fork}(\mathcal{N}'_k, \dots, \mathcal{N}'_m) \Rightarrow$$

    
$$l: \text{Fork}(\text{Minim}(\mathcal{N}'_1), \dots, \text{Minim}(\mathcal{N}'_k), \dots,$$

    
$$\text{Minim}(\mathcal{N}'_m), \dots, \text{Minim}(\mathcal{N}_n))$$

    
$$\text{Otherwise} \Rightarrow l: \text{Fork}(\text{Minim}(\mathcal{N}_1), \dots, \text{Minim}(\mathcal{N}_i), \dots, \text{Minim}(\mathcal{N}_n))$$

  
$$l: \text{Decision}((p_1, g_1, \mathcal{N}_1), \dots, (p_i, g_i, \mathcal{N}_i), \dots, (p_n, g_n, \mathcal{N}_n)) \Rightarrow \text{Case } (\mathcal{N}_i) \text{ of}$$

    
$$l: \text{Decision}((p'_1, g'_1, \mathcal{N}'_1), \dots, (p'_j, g'_j, \mathcal{N}'_j), \dots, (p'_m, g'_m, \mathcal{N}'_m)) \Rightarrow$$

    
$$l: \text{Decision}((p_1, g_1, \mathcal{N}_1), \dots, (p_i \times p'_1, g_i \wedge g'_1, \mathcal{N}'_1), \dots,$$

    
$$(p_i \times p'_j, g_i \wedge g'_j, \mathcal{N}'_j), \dots, (p_i \times p'_m, g_i \wedge g'_m, \mathcal{N}'_m),$$

    
$$\dots, (p_n, g_n, \mathcal{N}_n))$$

    
$$\text{Otherwise} \Rightarrow l: \text{Decision}((p_1, g_1, \text{Minim}(\mathcal{N}_1)), \dots, (p_n, g_n, \text{Minim}(\mathcal{N}_n)))$$

  
$$\text{Otherwise } \text{Minim}(\mathcal{N})$$


```

Listing 1.2. Control Nodes Abstraction Algorithm

resulting activity diagram has a reduced number of actions, which increases the occurrence of consecutive control nodes. Consequently, applying Abs first is more efficient³ as showed by the following proposition.

Proposition 1 (Application Order). *For a SysML activity diagram “ \mathcal{A} ” and a property “ ϕ ”, we have: $\text{Minim}(Abs(\text{Minim}(\mathcal{A}), \phi)) \equiv \text{Minim}(Abs(\mathcal{A}, \phi))$.*

Proof. Let $M_1 \equiv \text{Minim}(\mathcal{A})$, $M_2 \equiv Abs(M_1, \phi)$ and $M_3 \equiv \text{Minim}(M_2)$, we have:

1. $M_1 \equiv \text{Minim}(\mathcal{A}) \Leftrightarrow$ if $\exists l: \mathcal{N}_k \rightarrow \mathcal{N}_m \in \mathcal{A}$, then $l: \mathcal{N}_k \rightarrow \mathcal{N}_m$ is replaced by $l: \mathcal{N}_{km}$ if one of control merging rules is satisfied.
2. $M_2 \equiv Abs(M_1, \phi) \Leftrightarrow \forall a \notin \text{var}(\phi) : Abs(\overline{a}^n \rightarrow \mathcal{N}, \text{var}(\phi)) = \overline{a}^n \rightarrow \mathcal{N}$. In fact, Abs produces new consecutive control nodes and preserve the diagram structure.

It is clear that the second step has no effect on the first one and vice versa. In addition, applying Minim two times successively is equivalent to apply it once. Thus, the proposition holds. \square

5 Abstraction Soundness and Property Preservation

In this section, we first prove the soundness of our proposed abstraction algorithms. Next, we prove that our algorithms preserve the satisfaction of PCTL properties.

³ In term of time execution.

5.1 Abstraction Soundness

Our aim is to prove that our abstraction algorithm is sound and preserves PCTL properties. Let \mathcal{A} be a SysML activity diagram and $M_{\mathcal{A}}$ be its corresponding PA constructed by the NuAC operational semantics \mathcal{S} such that $\mathcal{S}(\mathcal{A}) = M_{\mathcal{A}}$. And, let δ be our abstraction composed of *Abs* and *Minim* algorithms such that $\delta(\mathcal{A}) = \widehat{\mathcal{A}}$, where $\widehat{\mathcal{A}}$ denotes the abstracted SysML activity diagram. Let $M_{\widehat{\mathcal{A}}}$ be its corresponding PA defined using the NuAC operational semantics \mathcal{S} such that $\mathcal{S}(\widehat{\mathcal{A}}) = M_{\widehat{\mathcal{A}}}$. As illustrated in Figure 4, proving the soundness of our algorithm is to find a relation \mathcal{R} between $M_{\mathcal{A}}$ and $M_{\widehat{\mathcal{A}}}$. The formal description of $M_{\mathcal{A}}$ is represented in Definition 1 where $Dist(S)$ is a convex distribution over a set S .

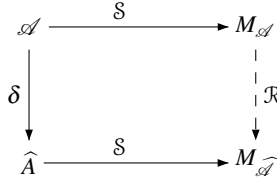


Fig. 4. Abstraction Correctness

Definition 1 (New Activity Calculus PA). A probabilistic automata of an activity calculus term \mathcal{A} is a tuple $M_{\mathcal{A}} = (\bar{s}, L, S, \Sigma, \delta)$ where:

- \bar{s} is an initial state, such that $L(\bar{s}) = \overline{\mathcal{A}}$,
- L is a labeling function,
- S is a finite set of states reachable from \bar{s} , such that, $S = \{s_i: 0 \leq i \leq n \mid L(s_i) \in \{\overline{\mathcal{N}}\}\}$,
- Σ is a finite set of actions corresponding to the alphabet of \mathcal{A} ,
- $\delta: S \times \Sigma \rightarrow Dist(S)$ is a (partial) probabilistic transition function such that, for each $s \in S$ and $\alpha \in \Sigma$ assigns a probabilistic distribution $\mu \in Dist(S)$ such that:
 - For each $S' \subseteq S$, $S' = \{s_i: 0 \leq i \leq n : s \xrightarrow{\alpha}_{p_i} s_i\}$. Each $s \xrightarrow{\alpha}_{p_i} s_i$ satisfies one NuAC semantic rule and $\mu(S') = \sum_{i=0}^n p_i = \sum_{i=0}^n \mu(s_i) = 1$.
 - For each transition $s \xrightarrow{\alpha}_{p_1} s''$ satisfying a NuAC semantic rule, μ is defined such that $\mu(s'') = 1$.

To define the relation $M_{\mathcal{A}} \mathcal{R} M_{\widehat{\mathcal{A}}}$, we introduce the notion of weak relation while δ abstracts away from invisible actions. Formally, the probabilistic weak simulation [23] relation defined in Definition 2 introduces the notion of observable action a preceded and followed invisible steps. We denote a weak transition by $(s \xRightarrow{a} P)$ where P is the distribution over states reached from s through a sequence of combined steps.

Definition 2 (Weak Probabilistic Simulation). A weak probabilistic simulation between two probabilistic automata M_1 and M_2 is a relation $\mathcal{R} \subseteq S_1 \times S_2$ such that:

1. each start state of M_1 is related to at least one start state of M_2 ,
2. for each pair of states $s_1 \mathcal{R} s_2$ and each transition $s_1 \xrightarrow{a} P_1$ of M_1 , there exist a weak combined transition $s_2 \xRightarrow{a} P_2$ of M_2 such that $P_1 \sqsubseteq_{\mathcal{R}} P_2$.

Here, $\sqsubseteq_{\mathcal{R}}$ is the lifting of \mathcal{R} to a probability space. It is achieved by finding a weight function [23] that associates each state of M_1 with others in M_2 by a certain probability value. It is defined below.

Definition 3 (Weight Function). A function $\Delta : S \times S' \rightarrow [0, 1]$ is a weight function for the two distribution $\mu_1, \mu_2 \in \text{Dist}(S)$ w.r.t. $\mathcal{R} \sqsubseteq S \times S'$ iff:

1. $\Delta(s_1, s_2) > 0 \Rightarrow (s_1, s_2) \in \mathcal{R}$,
2. $\forall s_1 \in S : \sum_{s_2 \in S} \Delta(s_1, s_2) = \mu_1(s_1)$,
3. $\forall s_2 \in S : \sum_{s_1 \in S} \Delta(s_1, s_2) = \mu_2(s_2)$ then $s_1 \mathcal{R} s_2$.

For our proof, we stipulate herein the abstraction relation denoted by $\mathcal{A} \sqsubseteq_{\mathcal{R}} \widehat{\mathcal{A}}$ between SysML activity diagrams \mathcal{A} and $\widehat{\mathcal{A}}$.

Definition 4 (Abstraction Relation). An abstraction relation is a weak probabilistic relation between a SysML activity diagram \mathcal{A} and its abstracted model $\widehat{\mathcal{A}}$ by applying δ algorithm.

In the following, we present the soundness of our algorithm. Let $\mathcal{M}_{\mathcal{A}}$ be a PA representing the semantic of the NuAC term \mathcal{A} , $\mathcal{M}_{\widehat{\mathcal{A}}}$ is the PA representing the semantics of $\widehat{\mathcal{A}}$ such that $\widehat{\mathcal{A}} = \delta(\mathcal{A}, \phi)$. Proving that δ is sound means proving there exists a weak probabilistic simulation between $\mathcal{M}_{\mathcal{A}}$ and $\mathcal{M}_{\widehat{\mathcal{A}}}$ i.e. $\mathcal{M}_{\mathcal{A}} \sqsubseteq_{\mathcal{R}} \mathcal{M}_{\widehat{\mathcal{A}}}$.

Theorem 1 (Soundness). The abstraction algorithm δ is sound.

Proof. The proof follows a structural induction on NuAC terms. In an inductive way, we select the $\bar{a} \mapsto \mathcal{N}$ case to prove the soundness for *Abs* procedure procedure. The remaining cases can be proved similarly for both *Abs* and *Minim* functions.

Let $L(s_1) = \bar{a} \mapsto \mathcal{N} \Rightarrow \exists s'_1 : s_1 \rightarrow s'_1$ by applying ACT-1 rule such that: $s'_1 = a \mapsto \mathcal{N} \Rightarrow \mu_1(s'_1) = 1$. By considering s_2 as the abstracted state of s_1 , $L(s_2) = \text{Abs}(L(s_1))$, we will have two cases:

1. $a \in \text{var}(\phi) : L(s_2) = \text{Abs}(\bar{a} \mapsto \mathcal{N}) = \bar{a} \mapsto \text{Abs}(\mathcal{N})$. By applying ACT-1, $\exists s'_2 : s_2 \rightarrow s'_2$ such that: $L(s'_2) = a \mapsto \text{Abs}(\mathcal{N}) \Rightarrow \mu_2(s'_2) = 1$. Then, it exists a weight function Δ for $\mathcal{R} = (s'_1, s'_2)$ such that:
 - (a) $\Delta(s'_1, s'_2) = 1 \Rightarrow \Delta(s'_1, s'_2) = \mu_1(s'_1)$, and
 - (b) $\Delta(s'_2, s'_1) = 1 \Rightarrow \mu_2(s'_2) = \Delta(s'_2, s'_1)$, then
 - (c) $\Delta(s_1, s_2) > 0 \Rightarrow s_1 \sqsubseteq_{\mathcal{R}} s_2$
2. $a \notin \text{var}(\phi) :$

$$L(s_2) = \text{Abs}(s_1) = \text{Abs}(\bar{a} \mapsto \mathcal{N}) = \bar{\varepsilon} \mapsto \text{Abs}(\mathcal{N}) \Rightarrow \exists s'_2 : s_2 \rightarrow s'_2.$$
 By applying ACT-1 rule such that $L(s'_2) = \varepsilon \mapsto \mathcal{N} \Rightarrow \mu_2(s'_2) = 1$.
 It exist a weight function Δ for $\mathcal{R} = (s'_1, s'_2)$ such that:
 - (a) $\Delta(s'_1, s'_2) = 1 \Rightarrow \Delta(s'_1, s'_2) = \mu_1(s'_1)$, and
 - (b) $\Delta(s'_2, s'_1) = 1 \Rightarrow \mu_2(s'_2) = \Delta(s'_2, s'_1)$, then
 - (c) $\Delta(s_1, s_2) > 0 \Rightarrow s_1 \sqsubseteq_{\mathcal{R}} s_2$

It is clear that, the marked NuAC term $\overline{\mathcal{A}}$ is the unique initial state of $M_{\mathcal{A}}$ corresponding to the unique initial state in $M_{\widehat{\mathcal{A}}}$. By following the same style of proof, we find:

$\overline{\mathcal{A}} \sqsubseteq_{\mathcal{R}} M_{\widehat{\mathcal{A}}}$, which confirms that Theorem 1 holds. \square

5.2 Property Preservation

In order to perform model-checking, a property should be specified. We selected PCTL to express such property. Formally, its syntax is given by the following BNF grammar:

$$\begin{aligned}\phi &::= \top \mid a \mid \phi \wedge \phi \mid \neg\phi \mid P_{\bowtie p}[\psi] \\ \psi &::= X\phi \mid \phi U^{\leq k}\phi \mid \phi U\phi\end{aligned}$$

Where a is an atomic proposition, $k \in \mathbb{N}$, $p \in [0, 1]$, and $\bowtie \in \{<, \leq, >, \geq\}$. Also, other useful operators can be derived such as:

- Future: $F\phi \equiv \top U \phi$ or $F^{\leq k}\phi \equiv \top U^{\leq k} \phi$.
- Generally: $G\phi \equiv \neg(F\neg\phi)$ or $G^{\leq k}\phi \equiv \neg(F^{\leq k}\neg\phi)$.

To specify a satisfaction relation of a PCTL formula in a state s , a class of adversaries (Adv) has been defined [8] to solve the nondeterminism decision. Hence, a PCTL formula should be satisfied under all adversaries. The satisfaction relation (\models_{Adv}) of PCTL formula is defined as follows:

- $s \models_{Adv} \top$
- $s \models_{Adv} a \Leftrightarrow a \in L(s)$
- $s \models_{Adv} \phi_1 \wedge \phi_2 \Leftrightarrow s \models_{Adv} \phi_1 \wedge s \models_{Adv} \phi_2$
- $s \models_{Adv} \neg\phi \Leftrightarrow s \not\models_{Adv} \phi$
- $s \models_{Adv} P_{\bowtie p}[\psi] \Leftrightarrow P(\{\pi \in IPath_{M,s} \mid \pi_{Adv} \models \psi\}) \bowtie p$
- $\pi \models_{Adv} X\phi \Leftrightarrow \pi(1) \models_{Adv} X\phi$
- $\pi \models_{Adv} \phi_1 U^{\leq k} \phi_2 \Leftrightarrow \exists i \geq k. (\pi(i) \models_{Adv} \phi_2 \wedge \pi(j) \models_{Adv} \phi_1 \forall j < i)$
- $\pi \models_{Adv} \phi_1 U\phi_2 \Leftrightarrow \exists k \geq 0. \pi \models_{Adv} \phi_1 U^{\leq k} \phi_2$

Here, we prove by induction on the structure of the PCTL grammar, except for the neXt operator ($PCTL_{\setminus X}$), that a formula (ϕ) holds in the concrete model if it holds in the abstracted model as stated in Theorem 2.

Theorem 2 (PCTL Preservation). *For two models M and \widehat{M} such that $M \sqsubseteq_{\mathcal{R}} \widehat{M}$. If ϕ is a $PCTL_{\setminus X}$ property, then we have: $(\widehat{M} \models \phi) \Rightarrow (M \models \phi)$.*

Proof. To prove the preservation of PCTL properties, we follow an inductive reasoning on the PCTL structure. \square

6 Experimental Results

In this section, we apply our abstraction algorithm on an online shopping system case study [25]. In order to show our abstraction efficiency, we use our translation algorithm [2] to map SysML activity diagrams into PRISM input language where we verify PCTL properties on both: the concrete and the abstract models. This is done in the purpose of providing experimental results demonstrating the efficiency and the validity of our abstraction.

To this end, we compare the results perspective of the verification cost (β) in terms of time verification and the abstraction efficiency (η) in terms of model's time construction. With respect to the verification cost, we measure both the time required to

construct the model, denoted by T_c , and the time required for verifying the property, denoted by T_v . The verification cost is given by $\beta = 1 - \frac{|T_v(\hat{M})|}{|T_v(M)|}$. Concerning the abstraction efficiency, we measure the number of transitions ($\#t$) for both concrete and abstract diagrams. It is given by $\eta = 1 - \frac{|T_c(\hat{M})|}{|T_c(M)|}$. The result of the verification of a property is denoted by Res .

6.1 Model Description

The online shopping system aims at providing services for purchasing online items. Figure 5a illustrates the corresponding SysML activity diagram⁴. It contains four call-behavior actions⁵, which are: “Browse Catalogue”, “Make Order”, “Process Order” and “Shipment”. As example, Figure 5b expands the call behavior action “Process Order”.

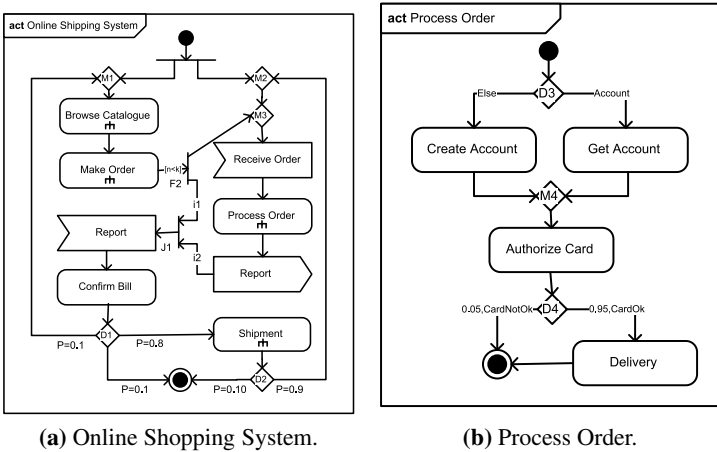


Fig. 5. The Concrete SysML Activity Diagram

6.2 Property Specification

In order to verify the functional requirements of the online shopping system, we propose the following properties and its related PCTL expressions.

1. For each order, what is the minimum probability value to make a delivery? From this expression, it is clear that only the main diagram (M) and “Process Order” behavior (M_3) are affected. We express this property in PCTL as follows, where n is the order number and K is the maximum allowed number to make an order.

$$Pmin = ?[(n \leq K) U (Delivery)]$$

⁴ This diagram is not symmetric which mean that we can not benefit from the symmetry reduction built within PRISM.

⁵ Each call-behavior action is represented by its proper diagram.

shows the evolution of the abstraction rate in terms of model size and computation time. In summary, the results demonstrate that the abstraction efficiency is important especially when the model' size is growing. Furthermore, they show that our abstraction algorithm actually preserves the verification results.

7 Conclusion

In this paper, we presented an automatic abstraction approach to improve the scalability of probabilistic model-checking in general and more especially for the verification of SysML activity diagrams. Also, we proposed a calculus dedicated to these diagrams. We have proved the soundness of our algorithm by defining a probabilistic weak simulation relation between the semantics of the abstract and the concrete models. In addition, the preservation of the satisfaction of $PCTL_{\setminus X}$ properties is proved. Finally, we demonstrated the effectiveness of our approach by applying it on an online shopping system application.

As future work, we would like to extend our approach by investigating several directions. First, we intend to integrate our algorithm within PRISM model checker. Second, we plan to apply our proposed abstraction on a composition of SysML activity diagrams. Next, we explore other abstraction approaches especially data abstraction targeting events and guards reduction. Finally, we intend to investigate reducing the property within the model at the same time to check $\widehat{M} \models \widehat{P}$ instead of $M \models P$.

References

1. Debbabi, M., Hassane, F., Jarraya, Y., Soeanu, A., Alawneh, L.: Verification and Validation in Systems Engineering - Assessing UML / SysML Design Models. Springer (2010)
2. Ouchani, S., Jarraya, Y., Mohamed, O.A.: Model-based systems security quantification. In: PST, pp. 142–149 (2011)
3. Baier, C., Katoen, J.P.: Principles of Model Checking, MIT Press (May 2008)
4. Xin-feng, Z., Jian-dong, W., Xin-feng, Z., Bin, L., Jun-wu, Z., Jun, W.: Methods to tackle state explosion problem in model checking. In: Proceedings of the 3rd Int. Conf. on IITA, pp. 329–331. IEEE Press, NJ (2009)
5. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
6. Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P.: Systems and Software Verification. Springer (2001)
7. OMG, OMG Systems Modeling Language (OMG SysML) Specification, Object Management Group, OMG Available Specification (September 2007)
8. Forejt, V., Kwiatkowska, M., Norman, G., Parker, D.: Automated Verification Techniques for Probabilistic Systems. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 53–113. Springer, Heidelberg (2011)
9. P. Team. PRISM - Probabilistic Symbolic Model Checker, <http://www.prismmodelchecker.org> (last visited November 2011)
10. Ober, I., Graf, S., Ober, I.: Model Checking of UML Models via a Mapping to Communicating Extended Timed Automata. In: SPIN 2004. LNCS, vol. 2989 (2004)
11. Ober, I., Graf, S., Ober, I.: Validating Timed UML models by simulation and verification. In: Workshop SVERTS, San Francisco (October 2003)

12. Westphal, B.: LSC Verification for UML Models with Unbounded Creation and Destruction. *Electronic Notes in Theoretical Computer Science* 144, 133–145 (2006)
13. Prashanth, C.M., Shet, K.C.: Efficient Algorithms for Verification of UML Statechart Models. *Journal of Software* 4, 175–182 (2009)
14. Daoxi, C., Guangquan, Z., Jianxi, F.: Abstraction framework and complexity of model checking based on the Promela models. In: 4th International Conference on Computer Science Education, ICCSE 2009, pp. 857–861 (July 2009)
15. Xie, F., Browne, J.C.: Integrated State Space Reduction for Model Checking Executable Object-Oriented Software System Designs. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 64–79. Springer, Heidelberg (2002)
16. Xie, F., Levin, V., Browne, J.C.: ObjectCheck: A Model Checking Tool for Executable Object-Oriented Software System Designs. In: *Fundamental Approaches to Software Engineering*, pp. 331–335 (2002)
17. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: A State/Event-Based Model-Checking Approach for the Analysis of Abstract System Properties. *Sci. Comput. Program.* 76, 119–135 (2011)
18. Yang, H.: Abstracting UML Behavior Diagrams for Verification. In: *Software Evolution With Uml And Xml*, pp. 296–320. IGI Publishing, Hershey (2005)
19. Eshuis, R.: Symbolic Model Checking of UML Activity Diagrams. *ACM Transactions on Software Engineering and Methodology* 15 (2006)
20. Eshuis, R., Wieringa, R.: Tool support for verifying uml activity diagrams. *IEEE Transactions on Software Engineering* 30 (2004)
21. Holt, J., Perry, S.: *SysML for Systems Engineering*. Institution of Engineering and Technology Press (January 2007)
22. OMG, *OMG Unified Modeling Language: Superstructure 2.1.2*, Object Management Group (November 2007)
23. Segala, R.: A Compositional Trace-Based Semantics for Probabilistic Automata. In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 234–248. Springer, Heidelberg (1995)
24. Milner, R.: *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press (1999)
25. Gomma, H.: *Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures*. Cambridge University Press (2011)

ML Dependency Analysis for Assessors

Philippe Ayrault^{1,2}, Vincent Benayoun³,
Catherine Dubois^{3,4,5}, and François Pessaux⁶

¹ Etersafe, Palaiseau, France

² Université Paris 6 - LIP6, Paris, France

³ CNAM - Laboratoire CEDRIC, Paris, France

⁴ ENSIIE, Evry, France

⁵ INRIA, Paris, France

⁶ ENSTA ParisTech - UEI, Palaiseau, France

Abstract. Critical software needs to obtain an assessment before commissioning. This assessment is given after a long task of software analysis performed by assessors. They may be helped by tools, used interactively, to build models using information-flow analysis. Tools like SPARK-Ada exist for Ada subsets used for critical software. But some emergent languages such as those of the ML family lack such adapted tools. Providing similar tools for ML languages requires special attention on specific features such as higher-order functions and pattern-matching. This paper presents an information-flow analysis for such a language specifically designed according to the needs of assessors. This analysis can be parametrized to allow assessors getting a view of dependencies at several levels of abstraction and gives the basis for an efficient fault tolerance analysis.

1 Introduction

Software is used to control everyday life as well as high risk systems. While games on smartphones and recreational instant messengers can contain software failures without staking human lives or implying catastrophic financial consequences, aeronautic trajectory controllers or automatic train protections are critical software.

Critical software must pass a safety assessment before its commissioning, given by sworn assessors. They perform a precise analysis as required by standards of the involved domains (IEC-61508 [16] for general purposes, CENELEC-50128 [15] for railways, etc.). This analysis aims at discovering any lack leading to feared events. Prior to software development, a hazard analysis states all safety prescriptions and drives software specification construction. Software analysis must convince the assessor that the development satisfies all safety requirements expressed in this specification. Assessment is a huge and difficult task, which amounts between 10 and 15% of the total development costs, due to the growing number of critical functionalities assigned to software components. Assessors can be helped by software analysis tools, however they do not want to delegate the

acceptance to a totally automatic and opaque tool, as they can be prosecuted in case of accident.

The assessment activity rests upon standard methodologies such as software *FMECA* [1] and *Fault Tree Analysis*, which are based on functional models. It mainly consists in exploiting some data flow analysis to build models, determine the impact of failures etc. Models construction especially relies on identification of the *real* inputs and outputs of a program or a software component (i.e. parameters, external entities – functions and constants – and side effects – accesses to values from terminals, files, sensors etc.) and *dependencies* between these inputs and outputs. Such models help recovering specifications of a software component, allowing to abstract it as a black box whose functional behaviour is simpler to manipulate. In the context of FMECA, the assessor has to study the effects resulting from the injection of failures on the functional behavior of a component. A fault injection is a modification of the value of an identifier (not necessarily erroneously), usually an input. As a consequence, we can also define a real input as an identifier “having an impact” on the component, i.e when it is modified (by a fault injection), the value of one of the outputs -at least- may change.

A last requirement in this kind of activity is to cope with abstraction. One may not be interested in the real dependencies of some components, either because we do not have their implementation or because they are considered “trusted” or out of scope. In such a case, we must be able to stop the analysis on such components, considering them as “terminal basic bricks” whose dependencies will be represented by the component’s name. Such components will be “tagged”, hence enabling a choice of the abstraction level. This choice is under the assessor’s responsibility, taking benefits of his experience in the domain of the system (e.g. railway) and his knowledge of this particular instance of it (e.g. a subway) he got through the documentation of the system. Determining the target of analysis then consists of understanding which components are considered critical and which ones can be safely abstracted.

Programming languages of the ML family become to be used in critical software development. Hence they need tools for computer-aided certification, in particular dependency analysis tools. Some industrial projects are already written using ML languages. Jane Street Capital develops critical trading systems in OCaml [9] which deals with hundreds of millions of dollars everyday. The certified embedded-code generator SCADE is also written in OCaml [12]. Many other critical software are developed using ML languages such as the Goanna static analysis for safety critical C/C++ [6], or the LexiFi Apropos software platform for pricing and management of financial products [8].

This paper presents a static analysis for functional programming languages to compute dependencies based on assessor’s needs, as recensed by one of the authors (P. Ayrault, an independent safety assessor (ISA) for railway domain) and informally presented in the previous paragraphs.

We first examine related works about information-flow analysis in functional languages in Section 2. In Section 3, the core language is introduced with its

¹ Failure Modes, Effects and Criticality Analysis.

syntax and operational semantics. Then Section 4 formalizes the dependency analysis aiming to address the previously quoted needs and states its correctness. This static analysis is the first step towards a tool enabling specification and verification of dependencies of a software component written in a functional language (like Spark-Ada does for Ada). This tool should allow the user to specify the real inputs and outputs, their dependencies and should be able to verify that the code is correct with respect to these specifications. In Section 5, we illustrate this on a small program. Then we conclude and give some possible extensions.

2 Related Work

Tools for model construction, based on information-flow analysis, exist for subsets of imperative languages. For example SPARK-Ada² gives the dependencies between inputs and outputs of software components written in an Ada subset. Functional languages begin to be used in development of critical software and tools related to safety. Unfortunately there are only very few tools dedicated to those languages and they are not as specifically designed for safety as are the tools for imperative languages.

Several motivations have led to information-flow analysis. The first one was compiler optimization with slicing [18], binding-time analysis [4,10], call-tracking [17] etc. Currently, the most studied matter is probably data security [7,14] (including secrecy and integrity).

Information-flow analysis for higher-order programming languages has been a long-term study.

In [2] the authors proposed a data-flow analysis for a lambda calculus. The goal of their analysis was run-time optimization using caching of previously computed values (do not compute an expression a second time if only independent values have changed). In their language, any sub-expression can be annotated with a label. Their run-time analysis is obtained by extending the usual operational semantics of the lambda calculus with a rule dealing with these labels. Hence, the analysis consists in evaluating the whole expression which leads to a value containing some of the labels present in the original expression. If a label is not present in the value, it means that the corresponding sub-expression is unused to compute the value.

In [1] the authors show that several kinds of information-flow analysis can be based on the same dependency calculus providing a general framework for secure information-flow analysis, binding-time analysis, slicing and call-tracking.

In [13] the analysis proposed by Abadi and al. in [2] is done statically using a simple translation and a standard type system. The authors claim that combined with the work of [1] their static analysis can also be used for all kinds of dependency analysis addressed in [1]. Later, in [14], the authors proposed a data-flow analysis technique for a lambda-calculus extended with references and

² See <http://libre.adacore.com/libre/tools/spark-gpl-edition/>

exceptions. A new approach based on a specific type system was proposed to fill the lacks of the previous approach.

This latter approach, used as a basis to build the Flow Caml language, uses a lattice representing security levels in order to ensure secrecy properties. As previously shown in [1], this kind of framework can be used for other kinds of dependency analysis by using different lattices. From our experiments, Flow Caml can be used to compute dependencies by “cheating”, “misusing” it and leads to pretty good results up to a certain point. The basic idea is to set one different security level per identifier to be traced, using a flat security lattice. Inference then mimics dependency analysis as long as no explicit type constraint with level annotation is present and as long as no references are used. This last point is the most blocking since invariance of references requires equality of levels. Hence, having different levels on the traced identifiers will make the type-checker rejecting such programs. In one sense, we need to be more conservative since we do not want to reject programs accepted by the “regular compiler”. We could say that Flow Caml accurately works but at some point, not in the direction we need. Moreover, the type system can’t “ignore” — i.e. really consider as opaque — dependencies for some identifiers the assessor doesn’t matter about. This is an important point since considering identifiers “abstract” means that the safety analysis must not take them into account for some reasons and allows reducing “noise” (flooding) polluting valuable dependency information. Assigning a convenient level on functions to trace them is to be more difficult since explicit type annotations get quickly complex. We succeeded for first order functions, even polymorphic, but it is unclear how to handle higher order.

All of these approaches try to provide an automatic information-flow analysis. This automation is valuable for the different purposes they address. However, assessors need a parametric tool that can be used interactively in order to help them building their own models.

3 The Core Language

3.1 Syntax

The language we consider is an extended λ -calculus with the most common features of functional programming languages: constants, sums, `let`-binding, recursion and pattern-matching. It comprises the functional kernel of CamlLight.

Sums are built from constant and parametrized constructors. Parametrized constructors only contain one parameter without loss of expressiveness since constructors with several parameters can be encoded using one parameter being a pair. In the same way, tuples can be encoded as nested pairs.

The considered pattern-matching only contains two branches where the second pattern is a variable, but it does not affect the language expressiveness since a pattern-matching with more branches can be encoded by nested pattern-matching with two branches. Conditional expressions are encoded as a pattern-matching against two constants constructors `True` and `False`.

Expressions

| | |
|---|---------------------------|
| $e ::= i$ | Integer constant |
| x | Identifier |
| $C \mid D(e) \mid (e_1, e_2)$ | Sum constructors and pair |
| $\mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2$ | let -binding |
| $\mathbf{fun} \ x \rightarrow e$ | Abstraction (function) |
| $\mathbf{rec} \ f \ x \rightarrow e$ | Recursive abstraction |
| $e_1 \ e_2$ | Application |
| $\mathbf{match} \ e_1 \ \mathbf{with} \ p \rightarrow e_2 \mid x \rightarrow e_3$ | Pattern-matching |

Patterns

| | |
|---------------|--------------------------|
| $p ::= x$ | Pattern variable |
| i | Integer constant pattern |
| $C \mid D(p)$ | Sum constructor patterns |
| (p_1, p_2) | Pair pattern |

Bindings

| | |
|------------------------------|-----------------------------|
| $s ::= \mathbf{let} \ x = e$ | <i>Toplevel let binding</i> |
|------------------------------|-----------------------------|

Programs (i.e. sequence of bindings)

| |
|--------------------|
| $prg ::= \epsilon$ |
| $s \ ; \ ; \ prg$ |

3.2 Operational Semantics

This section shortly presents the operational semantics of the language. This is a standard call-by-value semantics evaluating an expression to a value:

Values

| | |
|---|--|
| $v ::= i$ | Integer |
| $C \mid D(v)$ | Constant and parametrized constructors |
| (v_1, v_2) | Pair |
| $(\xi, \mathbf{fun} \ x \rightarrow e)$ | Simple closure (functional value) |
| $(\xi, \mathbf{rec} \ f \ x \rightarrow e)$ | Recursive closure |

where ξ is an evaluation environment mapping identifiers onto values.

Evaluation rules will be given for the 3 kinds of constructs of the language, i.e. expressions, patterns and definitions (which are mostly expressions bound at top-level and must be processed specifically since they largely contribute to the representation of dependencies, the bound identifiers being those appearing in the “visible” result). These rules are similar to those commonly presented for other functional languages.

Evaluation of expressions

$$\xi \vdash C \rightsquigarrow C \qquad \frac{\xi \vdash e \rightsquigarrow v}{\xi \vdash D(e) \rightsquigarrow D(v)}$$

Sum expressions are evaluated to the value corresponding to the constructor as introduced in its hosting type definition, embedding the values coming from the evaluation of their arguments if there are some.

$$\xi \vdash \mathbf{fun} \ id \rightarrow e \rightsquigarrow (\xi, \mathbf{fun} \ id \rightarrow e) \quad \xi \vdash \mathbf{rec} \ f \ id \rightarrow e \rightsquigarrow (\xi, \mathbf{rec} \ f \ id \rightarrow e)$$

Evaluation of a function expression leads to a “functional value” called a *closure*, embedding the current evaluation environment, the name of the formal parameter and the expression representing the body of this function.

$$\frac{\xi \vdash e_1 \rightsquigarrow (\xi_1, \mathbf{fun} \ id \rightarrow e) \quad \xi \vdash e_2 \rightsquigarrow v_2 \quad (id, v_2) \oplus \xi_1 \vdash e \rightsquigarrow v_3}{\xi \vdash e_1 \ e_2 \rightsquigarrow v_3}$$

$$\frac{\xi \vdash e_1 \rightsquigarrow (\xi_1, \mathbf{rec} \ f \ id \rightarrow e) \quad \xi \vdash e_2 \rightsquigarrow v_2 \quad (id, v_2) \oplus (f, (\xi_1, \mathbf{rec} \ f \ id \rightarrow e)) \oplus \xi_1 \vdash e \rightsquigarrow v_3}{\xi \vdash e_1 \ e_2 \rightsquigarrow v_3}$$

Evaluation of an application processes the argument expression, leading to a value v_2 and the functional expression leading to a closure. The environment of this closure is then extended (operator \oplus) by binding the formal parameter id to the value v_2 (and the function name to its closure in case of recursive function), then the body of the function gets evaluated in this local environment.

$$\frac{\xi \vdash e_1 \rightsquigarrow v_1 \quad (x, v_1) \oplus \xi \vdash e_2 \rightsquigarrow v_2}{\xi \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 \rightsquigarrow v_2}$$

The **let in** construct locally binds the identifier x to a value: the definition expression e_1 is first evaluated, let v_1 be its value, then the expression e_2 is evaluated in the extended environment binding the identifier x to the value v_1 .

$$\frac{\xi \vdash e_1 \rightsquigarrow v_1 \quad p, v_1 \vdash_p \xi_1 \quad \xi_1 \oplus \xi \vdash e_2 \rightsquigarrow v_2}{\xi \vdash \mathbf{match} \ e_1 \ \mathbf{with} \ p \rightarrow e_2 \mid x \rightarrow e_3 \rightsquigarrow v_2}$$

$$\frac{\xi \vdash e_1 \rightsquigarrow v_1 \quad \forall \xi_1. \neg(p, v_1 \vdash_p \xi_1) \quad (x, v_1) \oplus \xi \vdash e_3 \rightsquigarrow v_3}{\xi \vdash \mathbf{match} \ e_1 \ \mathbf{with} \ p \rightarrow e_2 \mid x \rightarrow e_3 \rightsquigarrow v_3}$$

Evaluation of a pattern-matching first processes the matched expression, leading to a value v_1 . We then need an extra operation, \vdash_p verifying that a value is “compatible” with a pattern and if so returns the bindings of pattern variables to add to the environment before the evaluation of the right-side expression of a matching case. The rules describing this operation follow below.

If the first case of the pattern-matching has a pattern, p , compatible with e_1 , then its induced pattern variables bindings are added to the current evaluation environment in which the right-side part expression of the case, e_2 , is evaluated resulting in the whole expression value.

If the first case does not match, then the second one, x , will always match, extending the evaluation environment by binding x to v in which the right-side part expression is evaluated.

Matching values against patterns

$$x, v \vdash_p (x, v) \quad C, C \vdash_p \emptyset \quad \frac{p, v \vdash_p \xi}{D(p), D(v) \vdash_p \xi} \quad \frac{p_1, v_1 \vdash_p \xi_1 \quad p_2, v_2 \vdash_p \xi_2}{(p_1, p_2), (v_1, v_2) \vdash_p \xi_1 \oplus \xi_2}$$

Evaluation of top-level definitions

$$\frac{\xi \vdash e \rightsquigarrow v}{\xi \vdash \mathbf{let} \ x = e \rightsquigarrow_s (x, v) \oplus \xi}$$

Evaluation of programs (successive top-level definitions)

$$\frac{\xi_0 \vdash \epsilon \rightsquigarrow_{prg} \xi_0 \quad \frac{\xi_0 \vdash s \rightsquigarrow_s \xi_1 \quad \xi_1 \vdash prg \rightsquigarrow_{prg} \xi_2}{\xi_0 \vdash s ; ; prg \rightsquigarrow_{prg} \xi_2}}{\xi_0 \vdash \epsilon \rightsquigarrow_{prg} \xi_0}$$

4 Dependency Analysis

The dependency analysis is described as an alternative semantics (like an abstract interpretation [5]) called “dependency semantics”. The dependency semantics performs a dependency inference by evaluating a program to a value (called dependency term) describing the dependencies of this program. This inference is the first step to verify dependencies stated by user specifications. The dependencies are expressed as structured values containing the “abstract” top-level identifiers having an impact on the evaluated program. External primitives or “trusted” components are given explicit dependencies since their definitions are either unavailable or unnecessary to analyze.

It is important to note that this analysis is more accurate than a “simple `grep`-like” command since the fact that an identifier does not appear in an expression does not mean that this expression does not depend on it. Moreover, the form of the dependency terms allows to keep trace of the structure of dependent data.

4.1 Dependency Semantics

The dependency on an identifier expression is simply described by the name of the identifier. A dependency term may be empty: a basic constant expression does not depend on anything. Constant and parametrized sum constructors as well as pairs allow representing the structure of the expression’s value. Union of dependencies allows combining dependencies of several sub-expressions.

Values: Dependency terms

| | |
|---------------------------------------|--|
| $\Delta ::= \perp$ | Empty dependency |
| x | Identifier (abstract or built-in) |
| $C \mid D(\Delta)$ | Constant and parametrized sum constructors |
| (Δ, Δ) | Pair |
| $\langle \lambda x.e, \Gamma \rangle$ | Closure |
| $\Delta \otimes \Delta$ | Union of dependencies |

Dependency semantics: Evaluation of expressions

The following judgments use a dependency environment Γ binding free identifiers to their corresponding dependency term.

$$(const) \quad \frac{}{\Gamma \vdash i \triangleright_e \perp}$$

A constant expression evaluates to an empty dependency: this means that the expression does not involve (depend on) any top-level identifier.

$$(ident) \quad \frac{}{\Gamma \vdash x \triangleright_e \Gamma(x)}$$

Evaluation of an identifier returns the value bound to this identifier in the environment. This value can be either the identifier itself, if the top-level definition of this identifier has been “tagged” as abstract, or another dependency term corresponding to its definition otherwise.

$$(pair) \quad \frac{\Gamma \vdash e_1 \triangleright_e \Delta_1 \quad \Gamma \vdash e_2 \triangleright_e \Delta_2}{\Gamma \vdash (e_1, e_2) \triangleright_e (\Delta_1, \Delta_2)}$$

A pair expression evaluates to a pair value in which each component is the evaluation of the corresponding component in the expression. This allows keeping trace of the dependency of each component separately. If only one of its components is used afterwards in the program, only the corresponding dependencies will be taken into account and not a significantly larger over-approximation.

$$(constr-1) \quad \frac{}{\Gamma \vdash C \triangleright_e C} \quad (constr-2) \quad \frac{\Gamma \vdash e \triangleright_e \Delta}{\Gamma \vdash D(e) \triangleright_e D(\Delta)}$$

Sum constructor expressions are evaluated to their dependency term counterpart, embedding the dependencies of their argument if they have some.

$$(lambda) \quad \frac{}{\Gamma \vdash \lambda x.e \triangleright_{e <} \lambda x.e, \Gamma >}$$

As in the operational semantics, functions evaluate to closures containing the **definition** of the function and the current environment. This rule has been chosen in order to provide a high level of precision, as opposed to the choice of analysing functions’ bodies to synthesize a term containing only partial information.

$$(recursion) \quad \frac{\Delta = \text{DepsOfFreeVars}(\lambda x.e, \Gamma)}{\Gamma \vdash \text{rec } f \ x.e \triangleright_{e <} \lambda x.e, (f, \Delta) \oplus \Gamma >}$$

where $\text{DepsOfFreeVars}(\lambda x.e, \Gamma)$ is the union (\otimes) of the dependency terms of all free variables present in the function definition. Γ contains the binding of each free variable to its dependency term, which has already been computed.

Recursive functions also evaluate to closures. As opposed to the operational semantics, there is no need of recursive closure. However to avoid forgetting dependencies caused by effective recursive calls, one must admit that recursive calls possibly depend on all free variables present in the body of the function. This is mandatory as the following example shows:

```
let abstr_id = ... ;; (* Assumed tagged ‘‘abstract’’ . *)
let rec f x = if x then f false else abstr_id ;;
let v = f true ;;
```

where omitting `abstr_id` which is free in the body of `f` is wrong since calling `f` with `true` involves a dependency during the recursive call.

Note that this approach is correct because **recursive calls** do not introduce extra dependencies except those coming from the evaluation of parameters and the body.

(apply-concr)

$$\frac{\Gamma \vdash e_1 \triangleright_e \Delta_1 \quad \Delta_1 = \langle \lambda x.e, \Gamma_x \rangle \quad \Gamma \vdash e_2 \triangleright_e \Delta_2 \quad (x, \Delta_2) \oplus \Gamma_x \vdash e \triangleright_e \Delta}{\Gamma \vdash e_1 e_2 \triangleright_e \Delta}$$

$$(apply-abstr) \quad \frac{\Gamma \vdash e_1 \triangleright_e \Delta_1 \quad \Delta_1 \neq \langle \lambda x.e, \Gamma_x \rangle \quad \Gamma \vdash e_2 \triangleright_e \Delta_2}{\Gamma \vdash e_1 e_2 \triangleright_e \Delta_1 \otimes \Delta_2}$$

There are two cases for the evaluation of an application. Either the function evaluates to a closure (see Rule *apply-concr*), hence the evaluation takes the same form as in the operational semantics or the function evaluates to something else (Rule *apply-abstr*) and then the dependencies are the approximation of the dependencies of both left and right expressions. This happens when a top-level function is tagged as “abstract”, (in this case the environment binds the identifier of the function to the dependency term reduced to this identifier, hence not revealing the dependency term of its body) or when the applied expression is not yet determined.

$$(let-in) \quad \frac{\Gamma \vdash e_1 \triangleright_e \Delta_1 \quad (x, \Delta_1) \oplus \Gamma \vdash e_2 \triangleright_e \Delta_2}{\Gamma \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 \triangleright_e \Delta_2}$$

The dependency semantics of a let binding mimics its operational semantics.

(match-static-1)

$$\frac{\Gamma \vdash e \triangleright_e \Delta \quad \mathit{statically_known}(\Delta) \quad p, \Delta \triangleright_m \Gamma_1 \quad \Gamma_1 \oplus \Gamma \vdash e_1 \triangleright_e \Delta_1}{\Gamma \vdash \mathbf{match} \ e \ \mathbf{with} \ p \rightarrow e_1 \mid x \rightarrow e_2 \triangleright_e \Delta_1}$$

(match-static-2)

$$\frac{\Gamma \vdash e \triangleright_e \Delta \quad \mathit{statically_known}(\Delta) \quad \forall \Gamma_1. \neg(p, \Delta \triangleright_m \Gamma_1) \quad (x, \Delta) \oplus \Gamma \vdash e_2 \triangleright_e \Delta_2}{\Gamma \vdash \mathbf{match} \ e \ \mathbf{with} \ p \rightarrow e_1 \mid x \rightarrow e_2 \triangleright_e \Delta_2}$$

During pattern-matching analysis, the structure of the matched value maybe statically known as expressed by the following predicate:

$$\mathit{statically_known}(\Delta) \triangleq \Delta = C \vee \Delta = D(_) \vee \Delta = (_, _)$$

In this case, we can then benefit from this information to deduce which branch to follow during the analysis as described by the two previous rules. The rule (*match-static-1*) applies when the dependency term of the matched expression has the same structure as the pattern of the first branch. If the first pattern does not match, the rule (*match-static-2*) applies. The \triangleright_m operation (whose rules are given below) computes the bindings induced by the pattern and the value and that must be added to the environment when analyzing each right-side part of the cases.

(match)

$$\frac{\Gamma \vdash e \triangleright_e \Delta \quad \neg(\mathit{statically_known}(\Delta)) \quad p, \Delta \triangleright_p \Gamma_1 \quad \Gamma_1 \oplus \Gamma \vdash e_1 \triangleright_e \Delta_1 \quad x, \Delta \triangleright_p \Gamma_2 \quad \Gamma_2 \oplus \Gamma \vdash e_2 \triangleright_e \Delta_2}{\Gamma \vdash \mathbf{match} \ e \ \mathbf{with} \ p \rightarrow e_1 \mid x \rightarrow e_2 \triangleright_e (\Delta \otimes \Delta_1 \otimes \Delta_2)}$$

When the branch is not statically known, an over-approximation of the real dependencies is done. The dependencies of the matched value are computed.

Then the expression of each branch is analyzed in the environment extended with the variables bound in the pattern (rules for \triangleright_p are given below). Finally the dependency of the whole expression is computed as the union of these three dependencies. Here, the structure of the value is lost because of the union of heterogeneous values.

Evaluation of definitions and programs

A top-level definition can be tagged as being either “abstract” or “concrete”. This choice influences which dependency term will be bound to the identifier being defined, hence allows selecting the level of abstraction, granularity of the analysis. In this way, the abstraction is directly controlled by the user and not through modifications of the analysis algorithm itself.

$$(let-concr) \quad \frac{\Gamma \vdash e \triangleright_e \Delta}{\Gamma \vdash \mathbf{let} \ x = e \triangleright_s (x, \Delta) \oplus \Gamma}$$

If the identifier is tagged as “concrete”, the rule (*let-concr*) applies and the identifier is bound to the dependency term resulting from the evaluation of its definition through the dependency semantics.

$$(let-abstr) \quad \frac{}{\Gamma \vdash \mathbf{let} \ x = e \triangleright_s (x, x) \oplus \Gamma}$$

Conversely, if the identifier is tagged as “abstract”, the rule (*let-abstr*) applies and the definition will bind the identifier to its name in the environment. This means that any use of this identifier in the program will be considered as depending only on this identifier and its own dependency term will be hidden.

$$\Gamma \vdash \epsilon \triangleright_{prg} \Gamma \quad (prog) \quad \frac{\Gamma_0 \vdash s \triangleright_s \Gamma_1 \quad \Gamma_1 \vdash prg \triangleright_{prg} \Gamma_2}{\Gamma_0 \vdash s ;; prg \triangleright_{prg} \Gamma_2}$$

The dependency semantics of a program mimics its operational semantics.

Evaluation of non-statically known pattern-matching

$$\frac{}{i, \Delta \triangleright_p \emptyset} \quad \frac{}{C, \Delta \triangleright_p \emptyset} \quad \frac{p, \Delta \triangleright_p \Gamma}{D(p), \Delta \triangleright_p \Gamma} \\ \frac{}{x, \Delta \triangleright_p (x, \Delta)} \quad \frac{p_1, \Delta \triangleright_p \Gamma_1 \quad p_2, \Delta \triangleright_p \Gamma_2}{(p_1, p_2), \Delta \triangleright_p \Gamma_1 \oplus \Gamma_2}$$

A pattern variable matches any dependency term and binds the corresponding identifier to the dependency term in the returned environment. Pattern-matching of pairs is done using an over-approximation because the matched dependency term is not necessary a pair (values considered as “abstract” can be matched).

Evaluation of statically known pattern-matching

$$\frac{}{x, \Delta \triangleright_m (x, \Delta)} \quad \frac{}{C, C \triangleright_m \emptyset} \\ \frac{p_1, \Delta_1 \triangleright_p \Gamma_1 \quad p_2, \Delta_2 \triangleright_p \Gamma_2}{(p_1, p_2), (\Delta_1, \Delta_2) \triangleright_m \Gamma_1 \oplus \Gamma_2} \quad \frac{p, \Delta \triangleright_p \Gamma}{D(p), D(\Delta) \triangleright_m \Gamma}$$

4.2 Proof of Correctness

The goal of the analysis is helping assessors to certify a given level of fault tolerance. In order to achieve this goal, the analysis must satisfy a correctness property based on the notion of fault tolerance. Hence, the notion of fault injection and its impact on the execution of a program are formalized and serve as a basis to express and prove a theorem of correctness.

Formalizing the notion of fault injection requires definitions of a reference environment (coming from the evaluation of the initial program) and an impact environment (from the evaluation after a fault injection). Similarity between those evaluation environments and the dependency environment (obtained from the dependency evaluation of the program) is defined below and serves as a basis to construct the proof of correctness of the analysis.

Definition 1 (Similar environments).

For a program $P = \text{let } x_1 = e_1, \dots, \text{let } x_n = e_n$ and an evaluation environment $\xi_n = (x_n, v_n), \dots, (x_1, v_1), \xi_\bullet$ (where ξ_\bullet is the environment corresponding to the free identifiers of the program), a dependency environment Γ_n is said similar to ξ_n if they share the same structure (i.e. $\Gamma_n = (x_n, \Delta_n), \dots, (x_1, \Delta_1), \Gamma_\bullet$) and if for any identifier x_i , the value v_i (resp. the dependency term Δ_i) corresponds to the operational (resp. dependency) evaluation of e_i in the environment ξ_{i-1} (resp. Γ_{i-1}).

Definition 2 (Fault injection on x_i).

Let $P = \text{let } x_1 = e_1, \dots, \text{let } x_n = e_n$ be a program. Injecting a fault in (P, x_i) at rank $i < n$ consists of building two environments ξ_{ref} (reference environment) and ξ_{impact} (impact environment) in the following way:

1. ξ_{ref} is the environment built during the evaluation of the program P .
2. If v_i is the value bound by the evaluation of the expression e_i (bound to x_i in the environment ξ_{ref}), then choose a value v different from v_i .
3. Build the environment ξ_{impact} by evaluating the rest of the program assuming that x_i is bound to the value v (instead of v_i):
 - for $j = 1 \dots i-1$, $\xi_{impact}(x_j) = \xi_{ref}(x_j)$.
 - $\xi_{impact}(x_i) = v$, fault injection on the identifier x_i by bypassing evaluation of expression e_i and binding x_i to v in the environment
 - for $j = i+1 \dots n$, $\xi_{impact}(x_j) = v_j$ with v_j being the result of the evaluation of the expression e_j : $(\xi_{impact_{j-1}} \vdash e_j \rightsquigarrow v_j)$.

Definition 3 (Impact of a fault injection).

Using the previous notations, given a program P and a reference environment ξ_{ref} we say that an expression e is impacted by the fault injection if there exist two different values v and v' and an impact environment ξ_{impact} obtained by a fault injection in (P, x_i) , such that $\xi_{ref} \vdash e \rightsquigarrow v$ and $\xi_{impact} \vdash e \rightsquigarrow v'$.

Theorem 1 (Correctness of the dependency analysis).

Let P be a program, e an expression of this program and x an identifier defined at top-level in P . Assuming that x is the only identifier tagged as “abstract” in P , if $\Gamma \vdash e \triangleright_e \Delta$ and x does not appear in Δ then e is not impacted by any fault injection in (P, x) .

Sketch of the proof. A reference evaluation environment is built by evaluating the program P through the operational semantics. An impact environment is obtained by a fault injection on x . Then a dependency environment is built by evaluating the program P through the dependency semantics. We then prove that the dependency and the evaluation environments are similar (according to the definition above).

A proof by induction on the dependency evaluation of the program states that in each case, the value of e is the same in the reference and the impact environments. For more details, a complete proof has been published in [3].

5 An Example

We now present a simple but relevant program to show how our dependency analysis could be used in practice. In the following, we display both top-level definitions and the corresponding results of the dependency analysis. We consider that the initial dependency environment contains some primitives like `+`, `-`, `abs`, `assert`, `fst`, `snd` with their usual meanings. This sample code depicts a simple voter system where 3 inputs are compared together with a given tolerance. The output is the most represented value and a validity flag telling if a given minimal number of inputs agreeing together has been reached. Obviously we are interested in the dependencies on the inputs and the two fixed parameters of the system: the threshold and minimal number of agreeing inputs. Inputs may be coming from the external environment, but for the analysis, since they are tagged “abstract”, they must be given each a dummy value.

The first five definitions introduce “abstract” identifiers. Their bodies being constants, their dependencies are \perp . However, since they are tagged “abstract”, identifiers are bound to a dependency denoting their name. Subsequent definitions bind functions, leading to dependencies being closures embedding the current environment. The interesting part mostly arises in the last definition where all the defined functions are applied, leading to a non-functional result. This result shows two kinds of information. First, we remark that the structure of the result is kept, i.e. is a pair, since the toplevel structure is (Δ_1, Δ_2) . Deeper structures (of Δ_1 and Δ_2) also exhibit the pair construct but are combined by \otimes , hence revealing that approximations of the analysis led to the loss of the knowledge of the real structure at this point. As long as a dependency term is not a union (\otimes), the analysis ensures that this term reflects the structure of the real value issued by effective computation. Conversely, apparition of \otimes dependencies stops guarantying this property. This means that in some cases, dependencies will still show all “abstract” identifiers an expression depends on, but won’t accurately show which part of the expression depends on which identifiers. The second interesting information is that, since the structure is kept in this case, the first component of the result does not depend on `min_quorum` whereas the second does.

```

(* The tagged ‘‘abstract’’ values, i.e. those to be traced. *)
(* Tolerance and minimal number of agreeing inputs. *)
let _threshold = 2 ;;           ▷e ⊥      Γ1 = (“_threshold”,_threshold) ⊕ Γ
let _min_quorum = 2 ;;        ▷e ⊥      Γ2 = (“_min_quorum”,_min_quorum) ⊕ Γ1

(* ‘‘Dummy’’ values set on inputs to perform the effective analysis. *)
let _sensor1 = 42 ;;          ▷e ⊥      Γ3 = (“_sensor1”,_sensor1) ⊕ Γ2
let _sensor2 = 45 ;;          ▷e ⊥      Γ4 = ...
let _sensor3 = 42 ;;          ▷e ⊥      Γ5 = ...

(* A comparison function modulo the threshold. *)
let eq v1 v2 =
  let delta = (v1 - v2) in
  (abs delta) <= _threshold ;;           ▷e < λ v1.λ v2.let delta = ...,Γ5 >

(* A validity check function ensuring that the minimum
   number of agreeing inputs is reached. *)
let valid num = num >= _min_quorum ;;   ▷e < λ num.(num ≥ _min_quorum),Γ6 >

(* The vote function returning the most represented value
   and the number of inputs agreeing on this value. *)
let vote in1 in2 in3 =
  let cmp1 = eq in1 in2 in
  let cmp2 = eq in2 in3 in
  let cmp3 = eq in3 in1 in
  match (cmp1, cmp2, cmp3) with
  | (false, false, false) -> (0, 4)
  | (true, true, false) | (true, false, true) | (false, true, true) ->
    assert false
  | (true, true, true) -> (in1, 3)
  | (true, _, _) -> (in1, 2)
  | (_, true, _) -> (in2, 2)
  | (_, _, true) -> (in3, 2) ;;
▷e < λ in1.λ in2.λ in3.let cmp1 = eq in1 in2 in...,Γ7 >

let main =
  let vot = vote _sensor1 _sensor2 _sensor3 in
  let response = fst vot in (* Get first component of the pair. *)
  let accordance = snd vot in (* Get the second one. *)
  let validity = valid accordance in
  (response, validity) ;;           ▷e
(
  ( (_threshold ⊗ _sensor2 ⊗ _sensor1,_threshold ⊗ _sensor3 ⊗ _sensor2),
    _threshold ⊗ _sensor1 ⊗ _sensor3 )
  ⊗ (_sensor1, ⊥) ⊗ (_sensor3, ⊥) ⊗ (_sensor2, ⊥)
,
  _min_quorum ⊗ (_sensor2, ⊥) ⊗ (_sensor3, ⊥) ⊗ (_sensor1, ⊥)
  ⊗
  ( (_threshold ⊗ _sensor2 ⊗ _sensor1,_threshold ⊗ _sensor3 ⊗ _sensor2),
    _threshold ⊗ _sensor1 ⊗ _sensor3 )
)

```

Future work addresses the problem of automatic verification of computed dependencies against those stated by the programmer. This implies being able to “understand” dependency terms. We explain here a few intuitions in this way. First of all, easy-to-read dependencies are those only containing identifiers or constructors. In this way, dependencies combined with a \otimes must be seen as unions of simple sets where \perp is the neutral element. Hence, $x_1 \otimes x_2 \otimes \perp$ means that dependency only implies identifiers x_1 and x_2 . When dependencies show constructors, we must look at identifiers appearing inside them. For instance, $A(x_1) \otimes B(x_2)$ also represents $\{x_1; x_2\}$. Difficulties arise with closure dependencies since no application reduced the function. A possible approach is digging in the term, harvesting both occurrences of arguments and identifiers

bound outside the function. This is a pretty crude approach but this still allows having information and in practice, we expect programs to be complete, i.e. with defined functions used, hence applied at some point. The loss of structure previously explained (see the code sample presentation) is also an issue since it makes more difficult checking the computed dependencies without structure against structured ones as the user could have stated them.

6 Conclusion and Further Work

Assessors perform huge tasks of manual software analysis. They may be assisted by automatic tools but they require tools specifically adapted for their needs and used interactively under their control. The dependency analysis presented in this paper has been implemented in a prototype (about 2000 lines of OCaml) and tested by an assessor on several examples such as a cruise control system. The adequacy of the analysis with higher-order functions has been demonstrated formally and experimentally. This tool answers a practical need of assessors by providing an information-flow analysis tool with a fine parametrization (using the tags `abstract/concrete`). This parametrization proved to be the backbone of the analysis.

Future work includes the development of the dependency verification tool as it was sketched on the example of Section 5. Next step is to produce a fine-grained analysis of pattern-matching based on the notion of execution paths. This more precise analysis would allow the assessor to recover or verify more accurately the dependencies between the real inputs/outputs, hence to get a finer model of the system under assessment.

Our analysis performs a kind of information-flow analysis sharing several aspects of the one done in Flow CAML. An extension of our analysis with notions present in Flow CAML (for instance security levels) may make our analysis able to check security properties along with the fault tolerance analysis.

Acknowledgement. We would like to express our greatest thanks to Thérèse Hardin for her participation in the work presented in this paper.

References

1. Abadi, M., Banerjee, A., Heintze, N., Riecke, J.G.: A core calculus of dependency. In: Appel, A.W., Aiken, A. (eds.) POPL, pp. 147–160. ACM (1999)
2. Abadi, M., Lampson, B.W., Lévy, J.-J.: Analysis and caching of dependencies. In: Harper, R., Wexelblat, R.L. (eds.) ICFP, pp. 83–91. ACM (1996)
3. Ayrault, P.: Développement de logiciel critique en Focalize. Méthodologie et outils pour l'évaluation de conformité. PhD thesis, Université Pierre et Marie Curie - LIP6 (2011)
4. Consel, C.: Binding time analysis for high order untyped functional languages. In: LISP and Functional Programming, pp. 264–272. ACM (1990)
5. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Aho, A.V., Zilles, S.N., Rosen, B.K. (eds.) POPL, pp. 269–282. ACM Press (1979)

6. Fehnker, A., Huuck, R., Jayet, P., Lussenburg, M., Rauch, F.: Goanna—A Static Model Checker. In: Brim, L., Haverkort, B.R., Leucker, M., van de Pol, J. (eds.) FMICS 2006 and PDMC 2006. LNCS, vol. 4346, pp. 297–300. Springer, Heidelberg (2007)
7. Heintze, N., Riecke, J.G.: The slam calculus: Programming with secrecy and integrity. In: MacQueen, D.B., Cardelli, L. (eds.) POPL, pp. 365–377. ACM (1998)
8. Jones, S.L.P., Eber, J.-M., Seward, J.: Composing contracts: an adventure in financial engineering, functional pearl. In: Odersky and Wadler [11], pp. 280–292
9. Minsky, Y., Weeks, S.: Caml trading - experiences with functional programming on wall street. *J. Funct. Program.* 18(4), 553–564 (2008)
10. Nielson, H.R., Nielson, F.: Automatic binding time analysis for a typed lambda-calculus. In: Ferrante, J., Mager, P. (eds.) POPL, pp. 98–106. ACM Press (1988)
11. Odersky, M., Wadler, P. (eds.): Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), Montreal, Canada, September 18-21. ACM (2000)
12. Pagano, B., Andrieu, O., Canou, B., Chailloux, E., Colaço, J.-L., Moniot, T., Wang, P.: Certified Development Tools Implementation in Objective Caml. In: Hudak, P., Warren, D.S. (eds.) PADL 2008. LNCS, vol. 4902, pp. 2–17. Springer, Heidelberg (2008)
13. Pottier, F., Conchon, S.: Information flow inference for free. In: Odersky and Wadler [11], pp. 46–57
14. Pottier, F., Simonet, V.: Information flow inference for ml. In: Launchbury, J., Mitchell, J.C. (eds.) POPL, pp. 319–330. ACM (2002)
15. Standard Cenelec EN 50128. Railway Applications - Communications, Signaling and Processing Systems - Software for Railway Control and Protection Systems (1999)
16. Standard IEC-61508, International Electrotechnical Commission. Functional safety of electrical/electronic/programmable electronic safety-related systems (1998)
17. Tang, Y.M., Jouvelot, P.: Effect systems with subtyping. In: Jones, N.D. (ed.) PEPM, pp. 45–53. ACM Press (1995)
18. Tip, F.: A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands (1994)

An Improved Test Generation Approach from Extended Finite State Machines Using Genetic Algorithms

Raluca Lefticaru and Florentin Ipate

Department of Mathematics and Computer Science, University of Pitești
Str. Târgu din Vale 1, 110040, Pitești, Romania
{Raluca.Lefticaru,Florentin.Ipate}@upit.ro

Abstract. This paper presents a new approach to test generation from extended finite state machines using genetic algorithms, by proposing a new fitness function for path data generation. The fitness function that guides the search is crucial for the success of a genetic algorithm; an improvement in the fitness function will reduce the duration of the generation process and increase the success chances of the search algorithm. The paper performs a comparison between the newly proposed fitness function and the most widely used function in the literature. The experimental results show that, for more complex paths, that can be logically decomposed into independent sub-paths, the new function outperforms the previously proposed function and the difference is statistically significant.

Keywords: fitness function, state-based testing, genetic algorithms.

1 Introduction

The continuous growth of software systems, in size and complexity, has increased the need for efficient and automated software testing. In recent years, software engineering, and in particular software testing, have known a novel approach, which transforms the software engineering task into an optimisation problem. This optimisation problem is then automatically solved using metaheuristic search techniques, such as Genetic Algorithms (GA), tabu search, ant colony or particle swarm optimisation. This new approach, namely Search-Based Software Testing (SBST) [13] has been studied by both industry and academy.

Metaheuristics have been used also for test generation from formal specification models, such as state machines [2,8,11]. This paper focusses on test generation for extended finite state machines (EFSMs) using genetic algorithms. Its main contribution is to propose a new fitness function for path data generation for EFSMs, that improves the previous function studied in the literature, in terms of success rates and efficiency of GA. The paper is structured as follows: section 2 briefly provides the background and section 3 presents the newly proposed fitness function. An empirical evaluation is realized in section 4. Finally, related work and conclusions are presented in sections 5 and 6.

2 Background

An *Extended Finite State Machine* (EFSM) is a 6-tuple (S, s_0, V, I, O, T) [8,17] where: S is a non empty set of logical states; $s_0 \in S$ is the initial state; V is the finite set of internal variables; I and O are the set of input and output interactions, respectively; T is the finite set of transitions. A *transition* $t \in T$ is represented by a 5-tuple (s_s, inp, g, op, s_e) in which: s_s and s_e represent the start state and the end state of t ; inp is the input, $inp \in I \cup \{Nil\}$, inp may have associated input parameters; g is the *guard* and is either *Nil* or is represented as a set of logical expressions given in terms of variables in V , parameters of the input interaction point and some constants; op is a computational block which consists of assignments and output statements. By *path* of an EFSM we mean a sequence of adjacent transitions of the EFSM [17]. In following, in order to facilitate the reading of longer transition paths, the input declarations considered will have the form $t_i(parlist)$ or $t_i()$ when the parameter list is empty, t_i representing a transition from T .

The problem we focus on in this paper is to generate test data for feasible paths in the state machine. A *feasible path* is a path for which there exist values for the input parameters, such as to satisfy all the guards and trigger all the transitions from the given path. However, the transition's computational block may assign to a variable a certain value, e.g. $x := 1$, and the next transition guard to check whether $x > 1$. A path containing these two successive transitions is *infeasible* because it is impossible to find input values to trigger it.

The problem of generating test data for feasible paths was studied in [8,11,12] and the solution proposed was to employ genetic algorithms or other metaheuristic search techniques, using a fitness function for state-based testing. An *individual* (or *chromosome*) is a list of input values, for example $x = (x_1, \dots, x_n)$, corresponding to all parameters of the path transitions (in the order they appear). The search spaces can be very large and solving the constraint system associated to the path is a NP-complete problem, which can be addressed with GA, for example. A fitness function, which evaluates the satisfaction of the guards (constraints) from the given path, and assigns better fitness values to the individuals that diverge later from the path was proposed in [11]. It is inspired by a well-known fitness function from structural testing, introduced by Wegener et. al [16], of the form: $fitness = approach_level + normalized_branch_level$. The *approach* (*approximation*) *level* is a metric that shows at which level the provided input determines the program to diverge from the target path [13]. The *branch level* estimates how close to being true the first unsatisfied condition is. Since the approach level takes only discrete values between $\{0, 1, \dots, m\}$, the branch level has to be normalized (mapped onto $[0; 1]$) [13]. The branch level is usually computed using the objective functions proposed by Korel [10] and further improved by Tracey in [15] (as described in Table 1). For normalization, exponential functions are usually employed, like $f(d) = 1 - 1.001^{-d}$, where $d \geq 0$ is the branch distance.

The formula $fitness = approach_level + normalized_branch_level$, for short $al + nbl$, was adapted for state-based testing in [11] and used also in [8] with a

Table 1. Traceys objective functions

| Relational predicate | Objective function |
|----------------------|--|
| $a = b$ | if $abs(a - b) = 0$ then 0 else $abs(a - b) + K$ |
| $a \neq b$ | if $abs(a - b) \neq 0$ then 0 else K |
| $a < b$ | if $a - b < 0$ then 0 else $(a - b) + K$ |
| Logical predicate | Objective function |
| Boolean | if $TRUE$ then 0 else K |
| $a \wedge b$ | $obj(a) + obj(b)$ |
| $a \vee b$ | $min(obj(a), obj(b))$ |

Require: Target path p , containing the transitions t_1, t_2, \dots, t_m , corresponding guards g_1, g_2, \dots, g_m , chromosome $x = (x_1, \dots, x_n)$

Ensure: The fitness value of the chromosome $x = (x_1, \dots, x_n)$ for the given path. Create an instance of the EFSM in the initial configuration.

```

approach_Level ←  $m - 1$ 
for  $i = 1 \rightarrow m$  do
  {for every transition  $t_i$  in the sequence  $p$ }
  if not  $g_i$  then
    Calculate  $obj(g_i)$ 
    return  $approach\_Level + norm(obj(g_i))$ 
  else
     $approach\_Level \leftarrow approach\_Level - 1$ 
    Apply transition  $t_i$  with the corresponding values from  $(x_1, \dots, x_n)$ 
  end if
end for
return 0
  
```

Fig. 1. Fitness function evaluation: $al + nbl$

slight modification. A high level description of the algorithm for computing the $al + nbl$ fitness function for state-based testing is given in Fig. 1, an illustrative example can be found in [11].

3 Fitness Function Based on Independent Sub-paths

In this paper we improve the above fitness by rewarding the individuals that satisfy more constraints from the path, even if they have diverged earlier. The conventional function $al + nbl$, which stops when a guard is violated, only takes into account the approach level and the local branch distance. Therefore, it does not make any use of information that is encoded in the chromosome after the place in which the first guard is violated. There are many cases when a guard does not evaluate any internal variables of the EFSM, e.g. $[x > 0]$ (or the guard evaluates some context variables, but these have not been modified since their initialization). Thus, this kind of constraint could be optimized "separately" and

then the values obtained for the particular transition could be used along with the other path values.

For example, consider the transition path $t_1(x_1, x_2) \rightarrow t_2(x_3, x_4) \rightarrow t_3(x_5, x_6, x_7) \rightarrow t_4(x_8)$ described below

| t | $s_s \rightarrow s_e$ | Input parameters | Transition guards | Transition actions |
|-------|-----------------------|----------------------|------------------------------------|---------------------------------|
| t_1 | $s_0 \rightarrow s_1$ | $t_1(x_1, x_2)$ | $g_1 : x_1 > 3 \wedge x_2 = 1$ | $v_1 := x_1; \quad v_2 := x_2;$ |
| t_2 | $s_1 \rightarrow s_2$ | $t_2(x_3, x_4)$ | $g_2 : x_3 = v_1 \wedge x_4 > v_2$ | $v_3 = x_3 - x_4;$ |
| t_3 | $s_2 \rightarrow s_3$ | $t_3(x_5, x_6, x_7)$ | $g_3 : x_5 > 0 \wedge x_6 > x_7$ | $v_1 = x_5 + x_6 - x_7;$ |
| t_4 | $s_3 \rightarrow s_4$ | $t_4(x_8)$ | $g_4 : x_8 = 10$ | $v_1 = x_8;$ |

If the transition t_1 is taken, then the internal variables v_1, v_2 are modified and then t_2 uses the modified v_1 and v_2 for the guard evaluation. This is the reason for the $al + nbl$ evaluation, that checks first if g_1 is satisfied and then if g_2 is true. On the other hand, the guards for the transition t_3 and t_4 , $[x_5 > 0 \wedge x_6 > x_7]$ and $x_8 = 10$, respectively, do not depend on the EFSM's previous configuration (internal variables). Considering that the path $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_4$ can be decomposed, at logical level, into *independent sub-paths* $[[t_1, t_2], [t_3], [t_4]]$, we propose an evaluation of the type $al + nbl$ for each sub-path and a global fitness function $path_fitness = \sum_{i=1}^{subpaths_no} fitness(subpath_i)$.

The difference between the conventional evaluation and the new one is that the latter will reward individuals based on the independent sub-paths, not only on the first block of genes from the chromosome, corresponding to the satisfied guards and the first unsatisfied condition.

| Falsified guards | Fitness value $al + nbl$ | Fitness value with independent components based-fitness |
|------------------|--------------------------|---|
| - | 0 | 0 |
| g_1 | $3 + norm(obj(g_1))$ | $(2 + norm(obj(g_1))) + 0 + 0$ |
| g_1, g_3 | $3 + norm(obj(g_1))$ | $(2 + norm(obj(g_1))) + (1 + norm(obj(g_3))) + 0$ |
| g_1, g_4 | $3 + norm(obj(g_1))$ | $(2 + norm(obj(g_1))) + 0 + (1 + norm(obj(g_4)))$ |
| g_2 | $2 + norm(obj(g_2))$ | $(1 + norm(obj(g_2))) + 0 + 0$ |
| g_2, g_3 | $2 + norm(obj(g_2))$ | $(1 + norm(obj(g_2))) + (1 + norm(obj(g_3))) + 0$ |
| g_3 | $1 + norm(obj(g_3))$ | $0 + (1 + norm(obj(g_3))) + 0$ |
| g_4 | $0 + norm(obj(g_4))$ | $0 + 0 + (1 + norm(obj(g_4)))$ |

Our intuition is that this new fitness function will behave the same as the conventional $al + nbl$ when the EFSM path has only one independent sub-path (in this case, each transition depends on at least one previous transition). Furthermore, we expect the new fitness function to give better results than the conventional one when the EFSM path has more independent sub-paths, because the level of satisfaction of each one is measured in this new formula. In what follows we will define the notion of independent sub-paths, show how these can be determined (Fig. 2) and finally present an algorithm for computing the new fitness function (Fig. 3). In the following definitions we will consider the EFSM model (S, s_0, V, I, O, T) .

1. For the given EFSM model build a transition-variable dependency matrix, *depends_on*, having the size $n \times m$, where n = number of transitions in EFSM, m = number of variables in V . $depends_on_{i,j} = 1$ if t_i depends on v_j and 0 otherwise.
2. Build a transition-variable modification matrix, *modifies*, having the size $n \times m$, $modifies_{i,j} = 1$ if t_i modifies the variable v_j and 0 otherwise.
3. Based on the previous matrixes and given a certain path $t_{I_1} \rightarrow \dots \rightarrow t_{I_p}$, compute a *transition-transition dependency* matrix, having the size $p \times p$, where p is the path length:

```

Initially dep has all the elements 0
for  $i = 2 \rightarrow p$  do
  for  $k = 1 \rightarrow m$  do
    if  $depends\_on_{I_i,k} = 1$  then
      {transition  $t_{I_i}$  depends on variable  $v_k$ }
      for  $j = i - 1 \rightarrow 1$  do
        if  $modifies_{I_j,k} = 1$  then
          {previous transition  $t_{I_j}$  modifies the variable  $v_k$ }
           $dep_{i,j} = 1$ ;
          {The  $i$ th transition from the path depends on the  $j$ th transition}
           $dep_{j,i} = 1$ ;
          {the matrix will be symmetric}
          break
          {no further dependencies are searched for  $t_{I_i}$  and  $v_k$ }
        end if
      end for
    end if
  end for
end for

```

4. Given this transition-transition dependency matrix *dep*, built for the path $t_{I_1} \rightarrow \dots \rightarrow t_{I_p}$ determine the connected graph components, or equivalent, the independent sub-paths.

Fig. 2. Computing the independent sub-paths

Definition 1. A transition $t \in T$, $t = (s_s, inp, g, op, s_e)$ depends on the internal variable $v_i \in V$ if the value of v_i is evaluated in the guard g .

Definition 2. A transition $t = (s_s, inp, g, op, s_e)$ modifies the internal variable $v_i \in V$ if v_i appears on the left hand side of an assignment operation of the *op* sequence of atomic operations.

Definition 3. Let $t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_i \rightarrow \dots \rightarrow t_j \rightarrow \dots \rightarrow t_p$ be a path in the EFSM. The transition $t_j = (s_s, inp, g, op, s_e)$ directly depends on the transition $t_i = (s'_s, inp', g', op', s'_e)$ if there exists an internal variable $v_k \in V$ such as t_i modifies the internal variable v_k , t_j depends on v_k and there is no other transition t_r , $i < r < j$ that modifies the internal variable v_k .

Discussion Regarding the ICF and al + nbl Functions

Normally, an EFSM transition is triggered when the guard is satisfied and the system is in the corresponding start state s_s for that transition. In the ICF

Require: Target path $t_{I_1} \rightarrow t_{I_2} \rightarrow \dots \rightarrow t_{I_p}$ containing the transitions to be triggered; corresponding guards $g_{I_1}, g_{I_2}, \dots, g_{I_p}$; set of independent subpaths of p (connected components in the dependency graph): $Indep = [[t_{J_{1,1}}, \dots, t_{J_{1,l_1}}], [t_{J_{2,1}}, \dots, t_{J_{2,l_2}}], \dots, [t_{J_{k,1}}, \dots, t_{J_{k,l_k}}]]$; chromosome $x = (x_1, \dots, x_n)$

Var $br_dist = (br_dist_1, \dots, br_dist_p)$ {array with branch distance for each guard}

Var $fit_comp = (fit_comp_1, \dots, fit_comp_k)$ {fitness of each indep. component}

begin

Instantiate the EFSM in the initial configuration

for $i = 1 \rightarrow p$ **do**

if not g_{I_i} **then**

$br_dist_i \leftarrow obj(g_{I_i})$

else

$br_dist_i \leftarrow 0$

end if

 Call the transition t_{I_i} , with the corresponding subset of values from (x_1, \dots, x_n) {If the current state of the EFSM is not the appropriate one, but the guard g_i is satisfied, the transition will be fired.}

end for

for $i = 1 \rightarrow k$ **do**

 {For each connected component compute the component fitness value}

$app_level \leftarrow l_i$ {initially approach level = length of the i th component}

$j \leftarrow 1$

repeat

if $br_dist_{t_{J_{i,j}}} > 0$ **then**

 {br.dist of the j th transition from i th independent component is > 0 }

$fit_comp_i \leftarrow app_level + norm(br_dist_{t_{J_{i,j}}})$

else

$app_level \leftarrow app_level - 1$

$j \leftarrow j + 1$

end if

until $j = l_i \vee br_dist_{t_{J_{i,j}}} > 0$

end for

return $fit_comp_1 + fit_comp_2 + \dots + fit_comp_k$

end

Fig. 3. Independent components-based fitness function (ICF)

evaluation, see Fig. 3 the current state is ignored, as it is not involved in the guard predicates. A drawback of the fitness function $al + nbl$ is the following: if some values from the chromosome satisfy the guard g_j , but some previous values did not satisfy their corresponding guard g_i , $i < j$, then evaluation does not take into account the g_j .

A transition can depend on several transitions. Therefore, the result of dependency relation is a directed graph, each node (vertex) representing a transition and each edge representing a dependence between two transitions. The dependency relation between transition, formulated in definition 3 takes into account only the last transition that modifies the variable. For example, if

$t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_4$ is a path in the EFSM, t_1 and t_3 modify the variable v_1 , the guard of t_4 evaluates v_1 , then we say that t_4 *directly depends on* t_3 and ignore the former dependence on t_1 . On the other hand, there are transitions that are clearly *independent* and form a sub-path containing a single transition. These transitions have guards that do not involve the internal variables from V , e.g. $x > 0$, where x is the input parameter of the transition, or, if the guards use a variable from V , then this variable has not been changed by any other previous transition from the given path. Having the *dependency graph* between transitions, one can easily obtain the connected components, using for example a classical algorithm [5]. The dependency graph is oriented and if t_i depends on t_j the reverse is never true. However, in order to easily obtain the connected components using the algorithm given in [5], we considered its adjacency matrix symmetric.

In the dependence relation constructed only the last transition that modifies a variable is taken into account and consequently some additional problems might appear. For example, if $t_1 \rightarrow t_2 \rightarrow t_3$ is a path in the EFSM, t_1 modifies a context variable v_1 , t_2 modifies other variable using the previously modified v_1 , e.g. $v_2 := v_1$ and t_3 evaluates in the guard the variable v_2 , then we conclude that t_3 depends on v_2 and furthermore that t_3 depends on t_2 . However, one can also trace back the dependencies and take into account that t_3 depends indirectly on t_1 .

4 Empirical Evaluation

4.1 EFSM Models

To compare the performances of the fitness functions presented before, we considered two EFSM models, given in Fig. 4. The first EFSM represents a library book and the second one a class 2 transport protocol. These two models were chosen because slightly modified versions of them were used in previous work on state-based testing, [11] and respectively [2,8,9].

The Book EFSM consists of four states, $S = \{s_0, s_1, s_2, s_3\}$, where s_0 is the initial state, a set of 2 internal variables $V = \{bId, rId\}$ (representing the costumers which borrowed and reserved the book, respectively) and 16 possible transitions. The model is self-explanatory, more details are provided below and in Table 2.

| State | Description | Context variables |
|-------|----------------------------|----------------------------------|
| s_0 | Book available | $bId = 0, rId = 0$ |
| s_1 | Book borrowed | $bId > 0, rId = 0$ |
| s_2 | Book reserved | $bId = 0, rId > 0$ |
| s_3 | Book borrowed and reserved | $bId > 0, rId > 0, bId \neq rId$ |

The second model is a simplified version of a class 2 transport protocol, that has been used in many experiments [2,8,9] and is considered nontrivial. This EFSM is based on the AP-module of the simplified transport protocol, for connecting to transport service access point and a mapping module respectively. It

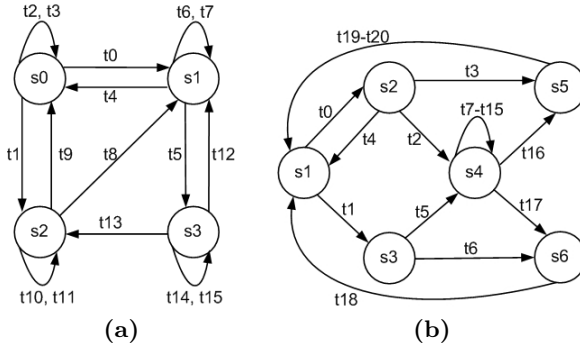


Fig. 4. EFSM models representing: (a) Book, (b) Class 2 transport protocol

consists of six states $\{s_1, s_2, s_3, s_4, s_5, s_6\}$, where s_1 is the initial state, five context variables $\{opt, R_credit, S_credit, TRsq, TSsq\}$ and 20 transitions. In Table 3, representing the transitions of the class 2 protocol, the output statements were ignored (as they did not modify the context variables) and in some cases the set of actions remained “Nil” after this deletion. The parameters written in italic are not used in assignment operations or in the guards and they were ignored for data generation, obtaining in this way an useful domain reduction.

It can be observed that the Book EFSM realizes assignment operations in which only parameters or constant values are assigned to the context variables, so it is no problem if the dependency relation described in section 3 takes into account only the direct dependencies. Furthermore, all the paths in the Book model are feasible. In the Protocol example, the assignments use also context variables on the right hand side, e.g.: $S_credit := S_credit - 1$ or $TSsq := (TSsq+1)mod128$. In this case, it would be useful to trace back the dependencies and find the transitions which previously modified the variables that appear on the right hand side of the assignment, in order to properly identify the independent sub-paths.

The Protocol example contains many infeasible paths, e.g. $t_1 \rightarrow t_5 \rightarrow t_{11} \rightarrow t_8$ and $t_0 \rightarrow t_2 \rightarrow t_{12} \rightarrow t_8$. In the given examples, both t_0 and t_1 execute the assignment $R_credit := 0$ which will falsify the guard of t_8 (containing the clause $R_credit \neq 0$), as no other modification of R_credit is realized on the given paths. Using a random path generation for Protocol paths of length 4 we obtained 15 such infeasible paths from 100 generated paths. For even longer transition sequences, the chances of obtaining an infeasible path are higher because: (a) transitions t_0 or t_1 will always appear (at least) as the first transition in every possible path and (b) it is very likely to have a t_8 transition and a definition clear sub-path for the variable R_credit , from t_0 or t_1 to t_8 .

4.2 Independent Sub-paths Examples

Given the following path in the Book EFSM: $t_1 \rightarrow t_8 \rightarrow t_7 \rightarrow t_4 \rightarrow t_0 \rightarrow t_4 \rightarrow t_0 \rightarrow t_5 \rightarrow t_{15} \rightarrow t_{12} \rightarrow t_4$, the independent sub-paths computed using the

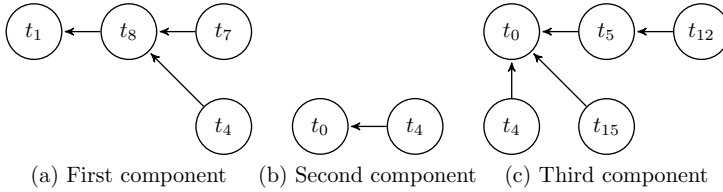
Table 2. The main transition of the Book EFSM

| t | $s_s \rightarrow s_e$ | Input | Description | Guards | Operations |
|----------|-----------------------|----------|---------------------------|---------------------------|-----------------------|
| t_0 | $s_0 \rightarrow s_1$ | $t0(x)$ | borrow book ok | $x > 0$ | $bId := x;$ |
| t_1 | $s_0 \rightarrow s_2$ | $t1(x)$ | reserve book ok | $x > 0$ | $rId := x;$ |
| t_2 | $s_0 \rightarrow s_0$ | $t2(x)$ | borrow book failed | $x \leq 0$ | Nil |
| t_3 | $s_0 \rightarrow s_0$ | $t3(x)$ | reserve book failed | $x \leq 0$ | Nil |
| t_4 | $s_1 \rightarrow s_0$ | $t4(x)$ | return book ok | $x = bId$ | $bId := 0;$ |
| t_5 | $s_1 \rightarrow s_3$ | $t5(x)$ | reserve book ok | $x > 0 \wedge x \neq bId$ | $rId := x;$ |
| t_6 | $s_1 \rightarrow s_1$ | $t6(x)$ | return book failed | $x \neq bId$ | Nil |
| t_7 | $s_1 \rightarrow s_1$ | $t7(x)$ | return book ok | $x \leq 0 \vee x = bId$ | Nil |
| t_8 | $s_2 \rightarrow s_1$ | $t8(x)$ | borrow book ok | $x = rId$ | $bId := x; rId := 0;$ |
| t_9 | $s_2 \rightarrow s_0$ | $t9(x)$ | cancel reservation ok | $x = rId$ | $rId := 0;$ |
| t_{10} | $s_2 \rightarrow s_2$ | $t10(x)$ | borrow book failed | $x \neq rId$ | Nil |
| t_{11} | $s_2 \rightarrow s_2$ | $t11(x)$ | cancel reservation failed | $x \neq rId$ | Nil |
| t_{12} | $s_3 \rightarrow s_1$ | $t12(x)$ | cancel reservation ok | $x = rId$ | $rId := 0;$ |
| t_{13} | $s_3 \rightarrow s_2$ | $t13(x)$ | return book ok | $x = bId$ | $bId := 0;$ |
| t_{14} | $s_3 \rightarrow s_3$ | $t14(x)$ | cancel reservation failed | $x \neq rId$ | Nil |
| t_{15} | $s_3 \rightarrow s_3$ | $t15(x)$ | return book failed | $x \neq bId$ | Nil |

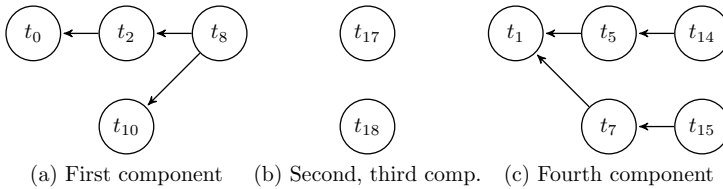
Table 3. The main transitions of the Class 2 transport Protocol (transitions $t_4, t_6, t_{16} - t_{20}$ are omitted because they have no guards nor assignment operations)

| t | $s_s \rightarrow s_e$ | Input declaration | Guards | Operations |
|----------|-----------------------|------------------------------------|---|--|
| t_0 | $s_1 \rightarrow s_2$ | $t1(dst_add, prop_opt)$ | Nil | $opt := prop_opt; R_credit := 0;$ |
| t_1 | $s_1 \rightarrow s_3$ | $t2(peer_add, opt_ind, cr)$ | Nil | $opt := opt_ind; S_credit := cr;$ $R_credit := 0;$ |
| t_2 | $s_2 \rightarrow s_4$ | $t2(opt_ind, cr)$ | $opt_ind < opt$ | $TRsq := 0; TSsq := 0;$ $opt := opt_ind; S_credit := cr;$ |
| t_3 | $s_2 \rightarrow s_5$ | $t3(opt_ind, cr)$ | $opt_ind > opt$ | Nil |
| t_5 | $s_3 \rightarrow s_4$ | $t5(accept_opt)$ | $accept_opt < opt$ | $opt := accept_opt; TRsq := 0;$ $TSsq := 0;$ |
| t_7 | $s_4 \rightarrow s_4$ | $t7(Udata, E0SDU)$ | $S_credit > 0$ | $S_credit := S_credit - 1;$ $TSsq := (TSsq + 1) \bmod 128;$ |
| t_8 | $s_4 \rightarrow s_4$ | $t8(Send_sq,$ $Ndata, E0TSDU)$ | $R_credit \neq 0$ AND $Send_sq = TRsq$ | $TRsq := (TRsq + 1) \bmod 128;$ $R_credit := R_credit - 1;$ |
| t_9 | $s_4 \rightarrow s_4$ | $t9(Send_sq,$ $Ndata, E0TSDU)$ | $R_credit = 0 \vee$ $Send_sq \neq TRsq$ | Nil |
| t_{10} | $s_4 \rightarrow s_4$ | $t10(cr)$ | Nil | $R_credit := R_credit + cr;$ |
| t_{11} | $s_4 \rightarrow s_4$ | $t11(XpSsq, cr)$ | $TSsq \geq XpSsq \wedge$ $cr + XpSsq - TSsq \geq 0 \wedge$ $cr + XpSsq - TSsq \leq 15$ | $S_credit := cr + XpSsq - TSsq;$ |
| t_{12} | $s_4 \rightarrow s_4$ | $t12(XpSsq, cr)$ | $TSsq \geq XpSsq \wedge$ $(cr + XpSsq - TSsq < 0 \vee$ $cr + XpSsq - TSsq > 0)$ | Nil |
| t_{13} | $s_4 \rightarrow s_4$ | $t13(XpSsq, cr)$ | $TSsq < XpSsq \wedge$ $cr + XpSsq - TSsq - 128 \geq 0 \wedge$ $cr + XpSsq - TSsq - 128 \leq 15$ | $S_credit := cr + XpSsq - TSsq$ $- 128;$ |
| t_{14} | $s_4 \rightarrow s_4$ | $t14(XpSsq, cr)$ | $TSsq < XpSsq \wedge$ $(cr + XpSsq - TSsq - 128 < 0$ $\vee cr + XpSsq - TSsq - 128 > 15)$ | Nil |
| t_{15} | $s_4 \rightarrow s_4$ | $t15()$ | $S_credit > 0$ | Nil |

algorithm from Fig. 2 are $[[t_1, t_8, t_7, t_4], [t_0, t_4], [t_0, t_5, t_{15}, t_{12}, t_4]]$ and the dependency graph has the following connected components:



Similarly, for the Protocol model, the independent components for the path $t_0 \rightarrow t_2 \rightarrow t_{10} \rightarrow t_8 \rightarrow t_{17} \rightarrow t_{18} \rightarrow t_1 \rightarrow t_5 \rightarrow t_{14} \rightarrow t_7 \rightarrow t_{15}$ are $[[t_0, t_2, t_{10}, t_8], [t_{17}], [t_{18}], [t_1, t_5, t_{14}, t_7, t_{15}]]$. This transition dependency graph below shows that the transition t_8 depends on two transitions t_2 and t_{10} , through the context variables $TRsq$ and R_credit , respectively.



4.3 Experiment Settings and Results

To compare the two fitness functions, we performed a controlled experiment, measuring the success rates and the efficiency of GA in the two cases and performing a statistical t-test to find out if the observed differences are statistically significant. In the experiment, we considered the two previously presented EFSM models and a large pool of randomly generated paths, for each model, having different lengths. For each randomly generated path, a genetic algorithm was applied 100 times using the conventional fitness function $al + nbl$ and again 100 times using the independent components-based fitness ICF . The performances of the GA in the two cases were recorded and analysed, with regard to the success rate and the average number of generations after the 100 runs. The null hypothesis (H_0) is thus formulated as follows: *There is no difference in efficiency (number of generations needed by the GA to find a solution) between the two fitness functions, $al + nbl$ and ICF .* The alternative hypothesis (H_a) is that there is a difference between the two fitness functions.

In the experiments we used the JGAP library [6]. The encoding for the Book model used integer-valued genes ranging in the domain $[-100, 100]$, the initialization of the initial generations was realized using uniformly distributed random numbers from this interval, starting each time from a different seed. The selection operator employed was *BestChromosomesSelector*, that takes the top 80% chromosomes into the next generation. Recombination was performed by means of the JGAP class *CrossoverOperator*, with the crossover rate fixed by default at $population_size/2$ [6], and mutation was realized using *MutationOperator* with a $1/12$ mutation rate. The population size was 20 individuals

and the maximum allowed number of evolutions was 1000 in all the cases. For statistical analysis the Apache Commons Mathematics Library was used <http://commons.apache.org/math/>. The statistical tests were realized using a significance level $\alpha = 0.05$, the p -values were also recorded, to decide if the tests were significant (confidence 95%) and very significant (confidence 99%). Due to space constraints, the detailed results are given in an Appendix, available at http://www.ifsoft.ro/~florentin.ipate/icf_results.pdf.

For the Book EFSM, a first set T_1 of 20 paths was generated, each with lengths between 6 and 15. The results obtained show that the new fitness function has a higher efficiency (the GA finishes in less generations) and the results are statistically significant for 14 out of 20 paths. Overall 65% of the differences turned out to be even very significant (13 out of 20 have the p -value much lower than 0.01, confidence 99%). The experiments showed that the difficulty of a path is not given only by its length (a longer path increases exponentially the search space), but also by the constraints that should be satisfied.

Additionally, we considered for the Book model two more test sets of 20 random paths each: T_2 , having paths of length 20, and T_3 , with paths of length 25. The purpose was to evaluate the fitness performances when the search space increases - in this case its dimension was 201^{20} and 201^{25} , respectively. For both test suites, *ICF* had better results than *al + nbl*, the differences were significant for 16 out of 20 paths in T_2 and for 18 paths out of 20 for T_3 . In the case of T_2 , *ICF* had better results in 15 out of 16 significant differences and, in the case of T_3 , in all of them. Furthermore, the results were very significant (99% confidence) for 15 paths in T_2 and 17 paths in T_3 , respectively. These results are summarized in Table 4.

As the complexity of the search problem increased, the GA ended in several cases without finding a solution. The results obtained for the paths of length 20 and 25 show that *ICF* also behaves better than the *al + nbl* function when the complexity of the search problem increases. Furthermore, the differences between the two functions are higher when the space increases.

A set of random paths was initially generated for the Protocol model, but compared with the Book EFSM, there are the following significant differences: (1) Many of the shorter paths (lengths 6-15) were *extremely easy to trigger* (usually test data was obtained from the first generation of the GA, by both fitness functions). (2) By increasing the path length to 25, 30 and 35, we obtained sets of randomly generated paths having a high percent of *infeasible paths* (approximately 50%). (3) The remaining generated paths were *easy to trigger* (the GA only needed a few generations, using either of the two fitness functions). Consequently, for most paths, the statistic tests could not reject the null hypothesis H_0 . To provide a convincing answer for our initial question, however, we increased the search space (each gene was coded using integer values from $[-500, 500]$) and studied the feasibility problem, in order to discard from the transition sets the great number of infeasible paths.

Consequently, we generated 60 longer paths, having lengths 25, 30 and 35, and we executed the GA 30 times for each path. For 32 paths, the GA could not

find a solution in any of the 30 runs. We manually inspected all these "difficult" paths and they were definitely infeasible. More precisely, all of them contained a t_8 transition (having the condition $R_credit \neq 0$ in the guard) after a t_0 or a t_1 call (which set $R_credit := 0$) and no intermediary transition to alter R_credit . On the other hand, paths that contain t_8 after a t_0 or t_1 can be feasible if intermediary transitions favourably change the value of R_credit (e.g. t_{10}). Using this property, we have modified the path generator such that, when a newly generated transition path contains t_8 but no t_{10} between the last call of t_0 or t_1 and t_8 , it replaces the t_8 transition with one that has the same start state and end state, by randomly choosing another transition from the set $\{t_7, t_9, t_{10}, \dots, t_{15}\}$.

Using this modified generator, we also increased the search space size in order to overcome the problem of "easy to trigger" paths; the following sets of feasible Protocol paths were generated: P_1 (20 paths with lengths between 6 and 15), P_2 (20 paths of length 20), P_3 (20 paths of length 25), P_4 (20 paths of length 30) and P_5 (20 paths of length 35).

After generating test data for the P_1 path set, we discovered that both fitness functions guided similarly the search because 15 out of 20 paths were *very easy to trigger* (in 1-2 generations) and 4 out of 20 paths were *easy to trigger* (in less than 100 generation, in which case the averages for the two functions were close and the p -value obtained was $0.10 \leq p \leq 0.40$). Only one path, having the independent components $[[t_0, t_3], [t_{20}], [t_1], [t_6], [t_{18}], [t_1, t_5, t_{10}, t_8, t_{11}], [t_{16}]]$, needed 593 generations for the GA with $al + nbl$ function and 505 generations for the *ICF*, but this difference was not considered statistically significant (the p -value was 0.11).

On the other hand, for the other 4 test sets, $P_2 - P_5$, by increasing the path length, the search problem became more difficult and we could observe several statistically significant differences between the two fitness functions, as presented in the Appendix available on-line. From a total of 23 paths rejecting the null hypothesis H_0 , the *ICF* outperformed the $al + nbl$ fitness in all 23 cases and the results were very significant for 15 out of 23 cases ($p < 0.01$). Note that 8 of these 23 paths are very easy to trigger (2-5 generations) and the other 15 are more complex. For the remaining paths from the sets $P_2 - P_5$, the performances of the two functions are comparable since, in general, these paths did not have a high complexity. In conclusion, in the case of the Protocol model, the results are similar for simpler paths. However, increasing the difficulty of the search problem, *ICF* outperforms $al + nbl$ and the scores obtained are statistically significant. Consequently, *ICF* is more adequately for more complex landscapes, when it can guide the search faster than the $al + nbl$ function.

The overall results obtained from this experiment, for both models, are summarized in Table 4. The first columns represent: the EFSM model used, the path set id, the number of paths in the test set and the length of the paths. The next three columns show: the number of cases in which the differences were statistically significant ($p < 0.05$), from these cases the number in which *ICF* is more efficient (ICF+) and in which $al+nbl$ is more efficient (Alnbl+), respectively.

Table 4. Summary of experimental results

| EFSM Model | Path Set | No. of Paths | Path Lengths | Stat. Signif. | ICF + | Alnbl + | Very Signif. | ICF + | Alnbl + |
|----------------|----------|--------------|--------------|---------------|-------|---------|--------------|-------|---------|
| Book | T1 | 20 | 6 – 15 | 14 | 14 | 0 | 13 | 13 | 0 |
| Book | T2 | 20 | 20 | 16 | 15 | 1 | 15 | 14 | 1 |
| Book | T3 | 20 | 25 | 18 | 18 | 0 | 17 | 17 | 0 |
| Book | T1–T3 | 60 | 6 – 25 | 48 | 47 | 1 | 45 | 44 | 1 |
| Protocol | P1 | 20 | 6 – 15 | 0 | 0 | 0 | 0 | 0 | 0 |
| Protocol | P2 | 20 | 20 | 6 | 6 | 0 | 3 | 3 | 0 |
| Protocol | P3 | 20 | 25 | 4 | 4 | 0 | 2 | 2 | 0 |
| Protocol | P4 | 20 | 30 | 6 | 6 | 0 | 4 | 4 | 0 |
| Protocol | P5 | 20 | 35 | 7 | 7 | 0 | 4 | 4 | 0 |
| Protocol | P1–P5 | 100 | 6 – 35 | 23 | 23 | 0 | 13 | 13 | 0 |
| Book, Protocol | All | 160 | 6 – 35 | 71 | 70 | 1 | 58 | 57 | 1 |

The last three columns have the same meaning, but refer to the very significant cases ($p < 0.01$).

Summarizing the results, as shown in Table 4, *ICF* function has a higher efficiency on 70 paths from 71 paths, on which the differences between functions were significant. The confidence in these tests is even higher for 58 cases, when the p -value was less than 0.01, and for these *ICF* obtained better scores in 57 out of 58 paths. The fitness function *ICF* obtained higher performances compared to the $al + nbl$ function, especially for more complex test paths.

The main threats to the validity of these empirical studies are: construct, internal and external validity threats. The first category includes the imprecision of cost measures such as number of iterations to make comparison between different search techniques. In our case, only GA were used, so this kind of measure was appropriate. Some internal validity threats can be: inadequate parameter settings for one or more of the search techniques (this was not the our case, involving only GA with same setting for fitness functions) or biased selection of the EFSM models, that have certain characteristics that can favour a certain fitness function.

The threats to external validity are the conditions that restrict our capacity to generalize these results. These can be related to the models used. Although the models are not trivial, using other EFSMs would give more confidence to the results. Furthermore, the length of the randomly generated paths could contribute differently to the performance of the fitness functions. As both types of paths were used: (a) with relatively short lengths (6-15) and (b) with a higher number of transitions (20, 25, 30, 35), we believe that this would not be a problem. However, this aspect should be considered in further research with additional EFSM case studies.

5 Related Work

Evolutionary approaches have been applied for different aspects of EFSM testing, such as: finding feasible transition paths and generating input sequences to

trigger these paths [1,3,7,8,9]; applying genetic algorithms in the case of timed extended finite state machines [2]; studying the efficiency of search based test generation for EFSM models [17].

Although our approach appears to be conceptually similar to the well-known Chaining approach [4], in that it takes into account the dependencies between variables, there are a number of key differences. First, while our method is devised to select test data from a model (EFSM), the chaining approach is based on a program (white-box testing). Furthermore, our algorithm uses a global, evolutionary inspired, search, whereas the chaining approach employs a local search (although attempts have been made to extend the technique to evolutionary testing [14]). Last, and most importantly, the chaining approach is aimed at determining test data to reach a given *target node* and, consequently, involves a complex process of constructing possible roots through the graph and eliminating those which are not feasible, whereas our method works on a *predefined path* and is aimed at increasing the success rate and efficiency of the algorithm which finds test data to execute the path.

6 Conclusions and Future Work

This paper proposes a new fitness function for path data generation from EFSMs, namely *independent component-based fitness* (ICF). This new function takes into account the independent sub-paths and it is expected to provide a better guidance to the search, compared to the conventional approach *al + nbl*. Its performance is evaluated for two EFSM models, Book and Protocol.

After performing statistic tests, we conclude that ICF obtains better results than *al + nbl* for the majority of the paths and it clearly improves the success rate of the GA, especially for complex paths. When the transition paths are easy to trigger, the two fitness functions have approximately the same behaviour and there is no significant difference between them. However, for complex paths the differences are very significant (confidence 99%) and the fitness function proposed clearly improves the conventional one.

As future work, we plan to improve the proposed approach by verifying the obtained results for more complex EFSMs, analysing how often independent sub-paths occur in real world EFSMs, comparing the results obtained by GA with other metaheuristic search techniques.

Acknowledgment. This work was supported by a grant of the Romanian National Authority for Scientific Research, CNCS–UEFISCDI, project number PN-II-ID-PCE-2011-3-0688.

References

1. Derderian, K., Hierons, R.M., Harman, M., Guo, Q.: Generating feasible input sequences for extended finite state machines (EFSMs) using genetic algorithms. In: GECCO 2005, pp. 1081–1082. ACM (2005)

2. Derderian, K., Merayo, M.G., Hierons, R.M., Núñez, M.: Aiding Test Case Generation in Temporally Constrained State Based Systems Using Genetic Algorithms. In: Cabestany, J., Sandoval, F., Prieto, A., Corchado, J.M. (eds.) IWANN 2009, Part I. LNCS, vol. 5517, pp. 327–334. Springer, Heidelberg (2009)
3. Derderian, K.A.: Automated Test Sequence Generation for Finite State Machines using Genetic Algorithms. Ph.D. thesis, School of Information Systems, Computing and Mathematics, Brunel University (2006)
4. Ferguson, R., Korel, B.: Software test data generation using the chaining approach. In: ITC, pp. 703–709. IEEE Computer Society (1995)
5. Hopcroft, J., Tarjan, R.: Algorithm 447: efficient algorithms for graph manipulation. *Commun. ACM* 16(6), 372–378 (1973)
6. Meffert, K. et al.: JGAP - Java Genetic Algorithms and Genetic Programming Package, <http://jgap.sf.net>
7. Kalaji, A., Hierons, R.M., Swift, S.: A search-based approach for automatic test generation from extended finite state machine (EFSM). In: TAIC-PART 2009, pp. 131–132. IEEE Computer Society (2009)
8. Kalaji, A., Hierons, R.M., Swift, S.: An integrated search-based approach for automatic testing from extended finite state machine (EFSM) models. *Information & Software Technology* 53(12), 1297–1318 (2011)
9. Kalaji, A., Hierons, R.M., Swift, S.: Generating feasible transition paths for testing from an extended finite state machine (EFSM). In: ICST 2009, pp. 230–239. IEEE Computer Society (2009)
10. Korel, B.: Automated software test data generation. *IEEE Trans. Softw. Eng.* 16(8), 870–879 (1990)
11. Lefticaru, R., Ipate, F.: Automatic state-based test generation using genetic algorithms. In: SYNASC 2007, pp. 188–195. IEEE Computer Society (2007)
12. Lefticaru, R., Ipate, F.: Functional search-based testing from state machines. In: ICST 2008, pp. 525–528. IEEE Computer Society (2008)
13. McMinn, P.: Search-based software test data generation: A survey. *Software Testing, Verification and Reliability* 14(2), 105–156 (2004)
14. McMinn, P., Holcombe, M.: Hybridizing Evolutionary Testing with the Chaining Approach. In: Deb, K., Tari, Z. (eds.) GECCO 2004. LNCS, vol. 3103, pp. 1363–1374. Springer, Heidelberg (2004)
15. Tracey, N., Clark, J., Mander, K., McDermid, J.: An automated framework for structural test-data generation. In: ASE 1998, pp. 285–288. IEEE Computer Society (1998)
16. Wegener, J., Baresel, A., Sthamer, H.: Evolutionary test environment for automatic structural testing. *Information & Software Technology* 43(14), 841–854 (2001)
17. Zhao, R., Harman, M., Li, Z.: Empirical study on the efficiency of search based test generation for EFSM models. In: ICST Workshops, pp. 222–231 (2010)

Securely Accessing Shared Resources with Concurrent Constraint Programming^{*,**}

Stefano Bistarelli^{1,2} and Francesco Santini^{1,3}

¹ Dipartimento di Matematica e Informatica, Università di Perugia, Italy
{bista, francesco.santini}@dmi.unipg.it

² Istituto di Informatica e Telematica, IIT-CNR, Pisa, Italy
stefano.bistarelli@iit.cnr.it

³ Centrum Wiskunde & Informatica, Amsterdam, Netherlands
F.Santini@cwi.nl

Abstract. We present a fine-grained security model to enforce the access control on the shared constraint store in Concurrent Constraint Programming (CCP) languages. We show the model for a nonmonotonic version of Soft CCP (SCCP), that is an extension of CCP where the constraints have a preference level associated with them. Crisp constraints can be modeled in the same framework as well. In the considered nonmonotonic soft version (NmSCCP), it is also possible to remove constraints from the store. The language can be used for coordinating agents on a common store of information that represents the set of shared resources. In such scenarios, it is clearly important to enforce the integrity and confidentiality rights on the resources, in order, for instance, to hide part of the information to some agents, or to prevent an agent to consume too many resources.

1 Introduction and Motivations

Coordination models and languages represent a very expressive approach for the development of applications for this class of dynamic and open systems such as operating systems, databases and mobile code. In the context of distributed/concurrent systems, the ability to coordinate the agents coupled with the possibility to control the actions they perform is significantly important. The necessity of guaranteeing security properties is rapidly arising in open and untrusted environments, due to several threats to the integrity and confidentiality properties of the exposed data.

* Work carried out during the tenure of an ERCIM "Alain Bensoussan" Fellowship Programme. This Programme is supported by the Marie Curie Co-funding of Regional, National and International Programmes (COFUND) of the European Commission.

** Research partially supported by MIUR PRIN 20089M932N project: "Innovative and multi-disciplinary approaches for constraint and preference reasoning", by CCOS FLOSS project "Software open source per la gestione dell'epigrafia dei corpus di lingue antiche", and by INDAM GNCS project "Fairness, Equità e Linguaggi".

The ingredient at the basis of our research is *Nonmonotonic Soft Concurrent Constraint Programming* (NmSCCP) [6]. The idea behind semiring-based soft constraints [4] was to further extend the classical constraint notion by adding the concept of a structure (i.e., a semiring) representing the levels of satisfiability of the constraints. The NmSCCP language extends the classical *Soft Concurrent Constraint Programming* (SCCP) language [5] with the possibility of relaxing (i.e., removing constraints) the store with a *retract* action, which clearly improves the expressivity of the language [6]. However, non-monotonicity raises further security concerns, since the store σ is a shared and centralized resource accessed in a concurrent manner by multiple agents at the same time: can an agent A relax a constraint c added to σ by the agent B ? Since in this case we are reasoning about soft constraints instead of crisp ones, “how much” of c can agent A relax? Security aspects related to constraint-based language have not been inspected yet in literature, even if foreboded in [12].

In this paper, we equip the core actions of the NmSCCP language [6] with a formal system of rights on the constraints, and then we study the execution of agents from this new perspective. We take inspiration from the *Access Control List* (ACL) model [18]. An ACL is a list of permissions attached to an object: it specifies which users or system processes are granted access to objects, as well as what operations are allowed on given objects.

In the proposed model, when an agent A_1 adds a piece of information to the store, i.e., a constraint c , it specifies also the confidentiality and integrity rights [21] on that constraint, for each agent A_i participating to the protected computation. For instance, how much of c the agent A_3 can remove from the store (i.e., the *retract rights*), or how much of c the agent A_2 can query with an *ask* operation (i.e., the *ask rights*). When an agent adds some information to the shared store (in a constraint-form), it also defines the ACL over that piece of information, via the same *tell* action. In our approach, both the resources in the store and the rights over them are represented as constraints. Supposing to know the number of agents at the beginning of the computation is a common practice in many security-related fields, as the execution of multiple threads on the same shared memory. We propose NmSCCP as a language to enforce a secure access over general shared resources, checking if quantitative rights over them are respected, e.g., “Peter may not eat more than 10% of the birthday cake”.

Moreover, as a further result with respect to [6,2], we introduce a new operation named *execp*, which can be used to execute a new agent in parallel; the inspiration comes from the *eval* operation of Linda (see Sec. 3). Our model can be easily specialized for crisp languages, as CCP [16], since crisp constraints can be represented in the same semiring-based framework as well (see Sec. 2).

The paper elaborates on a preliminary work [2] and it is organized as follows: in Sec. 2 we present the background information on soft constraints and the NmSCCP language. In Sec. 3 we describe the related work. In Sec. 4 we define the set of rights (for *tell*, *ask* and *retract* actions) on constraints, while in Sec. 5 we show the operational semantics of our secure language. Section 6 presents an example of secure coordination, and Sec. 7 draws the final conclusions.

$$\begin{aligned}
 c_i : (\{x\} \rightarrow \mathbb{N}) \rightarrow \mathbb{R}^+ \quad \text{s.t. } c_i(x) = 2x + 8 & \qquad c_j : (\{x\} \rightarrow \mathbb{N}) \rightarrow \mathbb{R}^+ \quad \text{s.t. } c_j(x) = x + 5 \\
 c_k : (\{x\} \rightarrow \mathbb{N}) \rightarrow \mathbb{R}^+ \quad \text{s.t. } c_k(x) = x + 3 & \qquad c_w : (\{x, y\} \rightarrow \mathbb{N}) \rightarrow \mathbb{R}^+ \quad \text{s.t. } c_w(x, y) = x + y + 5
 \end{aligned}$$

Fig. 1. Four weighted soft constraints evaluated $\langle \mathbb{R}^+ \cup \{+\infty\}, \min, \hat{+}, +\infty, 0 \rangle$

2 Soft Constraints and Nonmonotonic SCCP

Semiring-based constraints [4,3] rely on a simple algebraic structure, called “c-semiring” (simply semiring in the following), to formalize the notion of satisfaction/preference level associated with the constraint.

A **semiring** [3] S is a tuple $\langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ such that: *i*) A is a set and $\mathbf{0}, \mathbf{1} \in A$; *ii*) $+$ is commutative, associative and $\mathbf{0}$ is its unit element; *iii*) \times is associative, distributes over $+$, $\mathbf{1}$ is its unit element and $\mathbf{0}$ is its absorbing element. Moreover, $+$ is idempotent, $\mathbf{1}$ is its absorbing element and \times is commutative. Let us consider the relation \leq_S over A such that $a \leq_S b$ iff $a + b = b$. Then it is possible to prove that [4]: *i*) \leq_S is a partial order; *ii*) $+$ and \times are monotonic on \leq_S ; *iii*) $\mathbf{0}$ is its minimum and $\mathbf{1}$ its maximum; *iv*) $\langle A, \leq_S \rangle$ is a complete lattice and, for all $a, b \in A$, $a + b = \text{lub}(a, b)$ (where *lub* is the *least upper bound*). Informally, the relation \leq_S gives us a way to compare semiring values and then constraints. In fact, when we have $a \leq_S b$ we can say that b is *better than* a .

In [3] the authors extended the semiring structure by adding the notion of *division*, i.e., \div , as a weak inverse operation of \times for residuated semirings [3]. In this case, the set $\{x \in A \mid b \times x \leq a\}$ admits a maximum for all elements $a, b \in A$, denoted as $a \div b$. All classical soft constraint instances, i.e., *Classical* $\langle \{\text{false}, \text{true}\}, \vee, \wedge, \text{false}, \text{true} \rangle$, *Fuzzy* $\langle [0..1], \max, \min, 0, 1 \rangle$, *Probabilistic* $\langle [0..1], \max, \hat{\times}, 0, 1 \rangle$ and *Weighted* $\langle \mathbb{R}^+ \cup \{+\infty\}, \min, \hat{+}, +\infty, 0 \rangle$ (where $\hat{\times}$ and $\hat{+}$ respectively represent the arithmetic multiplication and addition), are residuated, and the notion of semiring division can be applied to all of them [3]. As an example, the inverse of the \times operator in $\langle \mathbb{R}^+ \cup \{+\infty\}, \min, \hat{+}, +\infty, 0 \rangle$ is represented by the arithmetic subtraction (i.e., $\hat{-}$), and it is defined as : $a \div b = \min\{x \mid b \hat{+} x \geq a\} = 0$ if $b \geq a$, or $a \hat{-} b$ if $a > b$.

A **soft constraint** [4] may be seen as a constraint where each instantiation of its variables has an associated preference. Given $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ and an ordered set of variables V over a (finite) domain D , a soft constraint is a function which, given an assignment $\eta : V \rightarrow D$ of the variables, returns a value of the semiring. Using this notation $C = \eta \rightarrow A$ is the set of all possible constraints that can be built starting from S, D and V . In Fig. 1 we show four weighted constraints (i.e., $\langle \mathbb{R}^+ \cup \{+\infty\}, \min, \hat{+}, +\infty, 0 \rangle$) as example.

Any function in C involves all the variables in V , but we impose that it depends on the assignment of only a finite subset of them. So, for instance, a binary constraint $c_{x,y}$ (as c_w in Fig. 1) over variables x and y , is a function

¹ The *Classical* semiring can be used to represent crisp constraints and, therefore, languages like CCP [16].

$c_{x,y} : (V \rightarrow D) \rightarrow A$, but it depends only on the assignment of variables $\{x, y\} \subseteq V$, that is the *support*, or *scope*, of the constraint. The function $supp : C \rightarrow V$ returns the variables that support a constraint. Note that $c\eta[v := d_1]$ means $c\eta'$ where η' is η modified with the assignment $v := d_1$. Note also that, with $c\eta$, the result we obtain is a semiring value, i.e., $c\eta = a$ with $a \in A$.²

Given the set C , the combination function of constraints $\otimes : C \times C \rightarrow C$ is defined as $(c_1 \otimes c_2)\eta = c_1\eta \times c_2\eta$. Considering Fig. 1, $c_k \otimes c_j = c_i$. Having defined the operation \div on semirings, the constraint division function $\ominus : C \times C \rightarrow C$ (which subtracts the second constraint from the first one) is instead defined as $(c_1 \ominus c_2)\eta = c_1\eta \div c_2\eta$ [3]. Considering Fig. 1, $c_i \ominus c_j = c_k$. Informally, performing \otimes or \ominus between two constraints means building a new constraint whose support involves all the variables of the original ones. This new constraint associates with each tuple of domain values for such variables a semiring element that is obtained by multiplying or, respectively, dividing the elements associated by the original constraints to the appropriate sub-tuples. According to their definition, the \otimes and \ominus operators respectively inherit the properties of \times and \div .

Given a constraint $c \in C$ and a variable $v \in V$, the *projection* [4] of c over $V - \{v\}$, written $c \downarrow_{(V \setminus \{v\})}$ is the constraint c' such that $c'\eta = \sum_{d \in D} c\eta[v := d]$. Informally, projecting means eliminating some variables from the support. This is done by associating with each tuple over the remaining variables a semiring element which is the sum of the elements associated by the original constraint to all the extensions of this tuple over the eliminated variables. Considering the examples in Fig. 1, $c_w \downarrow_{(\{y\})} = c_j$.

The partial order \leq_S over C can be easily extended among constraints by defining $c_1 \sqsubseteq_S c_2 \iff \forall \eta, c_1\eta \leq_S c_2\eta$; as an example, in Fig. 1 we have $c_j \sqsubseteq c_k$. Consider the set C and the partial order \sqsubseteq . The entailment relation $\vdash \subseteq \wp(C) \times C$ is defined s.t. for each $C \in \wp(C)$ and $c \in C$, we have $C \vdash c$ iff $\bigotimes C \sqsubseteq c$.

Nonmonotonic SCCP. With respect to classical *Concurrent Constraint Programming* (CCP) [16], in SCCP [5] the *tell* and *ask* agents are equipped with a preference (or consistency) threshold that is used to determine their success, failure, or suspension, as well as to prune the search.

The NmSCCP language [6] has an important difference with regard to the classical SCCP: the consistency level of the store can be increased by retracting constraints from it. For this reason, the semantics of the actions in NmSCCP includes also an upper bound on the store consistency, since it can be increased by a *retract*(c) action (for example). The *retract*(c) operation is at the basis of our nonmonotonic extension of the SCCP language, since it permits to remove the constraint c from the current store σ . It is worth to note that our *retract* can be considered as a “relaxation” of the store, and not only as a strict removal of the token representing the constraint, since in soft constraints we do not have the concept of token [4]. Thus, a constraint c can be removed even if it is different from any other constraints previously added to σ , as long it is entailed by the store, i.e., $\sigma \vdash c$.

² $\bar{0}$ and $\bar{1}$ respectively represent the constraints associating 0 and 1 to all assignments of domain values; in general, \bar{a} returns the semiring value a for all possible assignments.

$$\begin{array}{ll}
\mathbf{C1:} \ \succ \Rightarrow \xrightarrow{a_2}_{a_1} \text{ check}(\sigma)_{\succ} = \text{true} \text{ if } \begin{cases} \sigma \Downarrow_0 \not\leq_S a_2 \\ \sigma \Downarrow_0 \not\leq_S a_1 \end{cases} & \mathbf{C3:} \ \succ \Rightarrow \xrightarrow{\phi_2}_{\phi_1} \text{ check}(\sigma)_{\succ} = \text{true} \text{ if } \begin{cases} \sigma \Downarrow_0 \not\leq_S a_2 \\ \sigma \not\leq \phi_1 \end{cases} \\
\text{(with } a_1 \not\leq a_2) & \text{(with } \phi_1 \Downarrow_0 \not\leq a_2) \\
\mathbf{C2:} \ \succ \Rightarrow \xrightarrow{\phi_2}_{a_1} \text{ check}(\sigma)_{\succ} = \text{true} \text{ if } \begin{cases} \sigma \not\leq \phi_2 \\ \sigma \Downarrow_0 \not\leq_S a_1 \end{cases} & \mathbf{C4:} \ \succ \Rightarrow \xrightarrow{\phi_2}_{\phi_1} \text{ check}(\sigma)_{\succ} = \text{true} \text{ if } \begin{cases} \sigma \not\leq \phi_2 \\ \sigma \not\leq \phi_1 \end{cases} \\
\text{(with } a_1 \not\leq \phi_2 \Downarrow_0) & \text{(with } \phi_1 \not\leq \phi_2)
\end{array}$$

Otherwise, $\text{check}(\sigma)_{\succ} = \text{false}$

Fig. 2. Definition of the *check* function for each of the four checked transitions

Due to the non monotonic nature of the store, in [6] we have also added a *mask* operation, not present in SCCP, which is satisfied if a given constraint is *not* entailed by the current store.

In NmSCCP we also have an upper bound on the preference of the store, which is needed to prune “too good” computations obtained at a given step. In this way, we are able to model intervals of acceptability, while in SCCP there is only a check on “not good enough” computations, i.e., decreasing too much the consistency w.r.t the lower threshold. In Fig. 2 we show the new action prefixing symbol $\succ \Rightarrow$ in the syntax notation, which can be considered as a general “checked” transition of the type $\rightarrow_{\phi_1}^{\phi_2}$ (e.g., we can write $\text{ask}(c) \rightarrow_{\phi_1}^{\phi_2} A$), where ϕ_i is a placeholder that can stand for either a semiring element a_i or a constraint ϕ_i , i.e., $\phi_i ::= a_i | \phi_i$.

The syntax and the semantics of the core actions of NmSCCP are presented in Sec. 5 already enhanced with integrity and confidentiality rights and the new *execp* operation. In the following, we briefly describe an example of a parallel computation of the agents A_1 and A_2 , in order to familiarize the reader with NmSCCP (c_i, c_j and c_k are taken from Fig. 1).

$$A_1 \| A_2 :: \langle \langle \text{tell}(c_i) \rightarrow_0^{\bar{1}} \text{ask}(c_j) \rightarrow_0^{\bar{1}} \text{success} \rangle \parallel \langle \text{retract}(c_k) \rightarrow_0^{\bar{1}} \text{success} \rangle, \bar{1} \rangle$$

A computation starts in the empty initial store (i.e., $\bar{1}$). After A_1 adds c_i , then A_2 may remove c_k from the store and successfully terminate; after this, A_1 checks if c_j is still implied by the store (this holds), and then it may terminate in the *success* agent as A_2 . Since here we do not want to constrain the higher/lower preferences of the store, all the transitions are labeled with $\bar{0}/\bar{1}$ constraints.

3 Related Work

Most of the related proposals we report in this section come from studies on the Linda language, which is closely related to CCP [16]. Even if Linda has been extended by considering security mechanisms [20,10,19], as far as we know, no similar work has been predicted for the CCP language family, even if such scenario has been already predicted in [12].

The Linda model is implemented as a coordination language in which the primitives operate on an ordered sequence of typed data objects called “tuples”; these primitives are added to a sequential language, and interact with a logically global associative memory, called the tuplespace, in which processes store and retrieve tuples. The basic primitives are: **in**, which atomically reads and removes (i.e., consumes) a tuple from tuplespace, **rd** non-destructively reads a tuplespace, **out** produces a tuple, writing it into tuplespace, and **eval** creates new processes to evaluate tuples, writing the result into tuplespace.

In [20] the authors present *SecSpaces*, the design of a new coordination model which extends Linda with fine-grained access control; *SecSpaces* is a capability-based system where the dynamic privileges acquisition happens only if an agent reads an entry that contains a control field value in the data fields.

In [10] the authors present a novel coordination model which provides three mechanisms to manage tuple access control. The first mechanism supports logical partitions of the shared repository: in this way it is possible to restrict the access to tuples inside a partition, simply by limiting the access to the partition itself. The second mechanism consists of adding to the tuples some extra information that exploits asymmetric cryptography, in order, for instance, to authenticate the producer of a tuple or to identify its reader/consumer. Finally, there is support to define access-control policies based on the kind of operations an agent performs on a tuple, thus discriminating between (destructive) input and (non-destructive) read permissions on each single tuple.

One interesting proposal is the *Klaim* language [14]. It exploits a standard access control mechanism where permissions describe which operations the agents can perform on the available spaces. In its original version, *Klaim* does not allow dynamic permission acquisition; this is supported by *MuKlaim* [9], that is a sort of core for *Klaim*, since it implements its basic features. Further, the solution offered by *Klaim* is not particularly suitable for environments where the system configuration changes frequently.

Other related proposals are *Klava* [1] and a secure version of *Lime* [11]. The former introduces encrypted messages into the fields of the tuples, and the matching rule allows the evaluation of messages encrypted into fields; the encryption of messages ensures that they can be read only by the allowed clients. In [11] the authors present a secure implementation of *Lime* (*Linda in Mobile Environment*), that provides a password-based access control mechanism at the level of tuples and tuple spaces. Finally, in [15] the authors propose a general model for coordination middlewares that exploits process handlers to control the behaviour of processes; a language derived from CCS is used to describe which operations are allowed.

Comparing the previous works based on Linda with our solution for NmSCCP, the main difference consists in the fact that, in our case, the controls cannot be applied on a single piece of information, as a token in crisp constraint systems or tuples in the Linda store. The reason is that in a semiring-based framework the information is mixed in the store without any identity, and it is represented by quantities that can be even partially removed from it.

4 Defining the Tell/Ask/Retract Rights

In the language presented in this paper (see Sec. 5), we use control mechanisms in order to guarantee integrity and confidentiality [21] on a shared store of constraints. However, since we work on a semiring-based formalism (see Sec. 2), our checks need to be focused on a quantitative, rather than qualitative, point of view, differently from previous works on Linda (see Sec. 3). In fact, our approach is able to set “how much” of the current store each agent can *retract* or *ask*. Therefore, also the rights, together with the information they are applied on (i.e., soft constraints) are soft, in the sense they may also concern part of the added information. In a crisp vision of access control, if c_1 is added to the store, it would be possible to prevent only the removal of the entire c_1 , but not part of it as, instead, we can manage with NmSCCP.

When an agent inserts a constraint into the store by performing a *tell* action, it also specifies the rights that all the other agents have on that constraint (see Sec. 5 for the semantics of the operation). We define three kinds of rights: the *tell rights*, stating how much the added constraint can be “worsened” by the other agents, the *ask rights*, which specify how much of the constraint can be “read” by each agent, and the *retract rights*, describing how much of the added constraint can be removed via a *retract* action. This approach is inspired by the classical matrix-based ACL security model [18]. Each entry in a typical ACL specifies a subject and an operation and follows the structure: (object identity, user identity) \rightarrow permitted operations. The *tell* and *retract rights* can be classified as *integrity rights* [21], while the *ask rights* are classified as *confidentiality rights* [21].

Definition 1 (Tell, Ask and Retract Rights)

Tell Rights: Each constraint c_k added to the store is associated with a vector $\mathfrak{R}_t = \langle c_{t_1}, c_{t_2}, \dots, c_{t_n} \rangle$, where n is the number of agents participating to the concurrent computation. c_{t_i} represents the tell right imposed on agent A_i . In particular, c_{t_i} represents how much of c_k the agent A_i can write in the store with successive tell operations, that is, how much A_i can worsen c_k .

Ask Rights. Each constraint c_k added to the store is associated with a vector $\mathfrak{R}_a = \langle c_{a_1}, c_{a_2}, \dots, c_{a_n} \rangle$, where n is the number of agents participating to the concurrent computation. c_{a_i} represents the ask right imposed on agent A_i . In particular, c_{a_i} represents how much of the added c_k constraint can be read (with an ask operation) by agent A_i .

Retract Rights: Each constraint c_k added to the store is associated with a vector $\mathfrak{R}_r = \langle c_{r_1}, c_{r_2}, \dots, c_{r_n} \rangle$, where n is the number of agents participating to the concurrent computation. c_{r_i} represents the retract rights imposed on agent A_i , that is, how much of c_k can be removed by agent A_i with a retract operation.

The three right vectors \mathfrak{R}_t , \mathfrak{R}_a and \mathfrak{R}_r are represented in Tab. 1 as a matrix of rights \mathfrak{R} , which collects the ACL for each entity and for each one of the three kinds of right. This matrix of rights represents a new kind of information for CCP-like languages, and it belongs to the state of the computation of NmSCCP agents: it is updated at each step via the NmSCCP actions (see Sec. 5).

Table 1. The vision of the store for the imposed rights given as the \mathfrak{R} matrix

| | \mathfrak{R}_t (Tell) | \mathfrak{R}_a (Ask) | \mathfrak{R}_r (Retract) |
|-----------|-------------------------|------------------------|----------------------------|
| $Agent_1$ | $\mathfrak{R}_t[1]$ | $\mathfrak{R}_a[1]$ | $\mathfrak{R}_r[1]$ |
| $Agent_2$ | $\mathfrak{R}_t[2]$ | $\mathfrak{R}_a[2]$ | $\mathfrak{R}_r[2]$ |
| ... | ... | ... | ... |
| $Agent_n$ | $\mathfrak{R}_t[n]$ | $\mathfrak{R}_a[n]$ | $\mathfrak{R}_r[n]$ |

With an abuse of notation we define the composition of rights operation as $\mathfrak{R}' = \mathfrak{R} \otimes \bar{\mathfrak{R}}$, where \mathfrak{R} and \mathfrak{R}' respectively model the former and the new matrix of rights in the computation state, while $\bar{\mathfrak{R}}$ represents the new rights that modify the former matrix. As already anticipated, $\bar{\mathfrak{R}}$ is a parameter of the *tell* action of our language (see Fig. 3). $\mathfrak{R}' = \mathfrak{R} \otimes \bar{\mathfrak{R}}$ is implemented with equations (1), (2) and (3) (i.e., respectively *tell*, *ask* and *retract rights*): (1) $\forall i. \mathfrak{R}'_t[i] = \mathfrak{R}_t[i] \otimes \bar{\mathfrak{R}}_t[i]$, (2) $\forall i. \mathfrak{R}'_a[i] = \mathfrak{R}_a[i] \otimes \bar{\mathfrak{R}}_a[i]$, (3) $\forall i. \mathfrak{R}'_r[i] = \mathfrak{R}_r[i] \otimes \bar{\mathfrak{R}}_r[i]$. Note that we use the \otimes (see Sec. 2) because even rights are soft constraints.

As an example of composition of rights, we consider the *Weighted* semiring $\langle \mathbb{R}^+ \cup \{+\infty\}, \min, +, +\infty, 0 \rangle$ and the following \mathfrak{R} and $\bar{\mathfrak{R}}$:

$$\mathfrak{R} = (\mathfrak{R}_t = \langle x, \bar{5}, x + y \rangle, \mathfrak{R}_a = \langle y, x, \bar{1} \rangle, \mathfrak{R}_r = \langle x, z, \bar{2} \rangle)$$

$$\bar{\mathfrak{R}} = (\bar{\mathfrak{R}}_t = \langle y, x, \bar{3} \rangle, \bar{\mathfrak{R}}_a = \langle \bar{1}, \bar{1}, \bar{1} \rangle, \bar{\mathfrak{R}}_r = \langle \bar{1}, x, \bar{6} \rangle)$$

the \mathfrak{R}' composition of rights is given by

$$\mathfrak{R}' = \mathfrak{R} \otimes \bar{\mathfrak{R}} = (\mathfrak{R}'_t = \langle x + y, x + \bar{5}, x + y + \bar{3} \rangle, \mathfrak{R}'_a = \langle y, x, \bar{1} \rangle, \mathfrak{R}'_r = \langle x, x + z, \bar{8} \rangle)$$

Note that, since rights on soft constraints are represented by soft constraints as well, they behave in the same way: when new rights are added to the security matrix ($\mathfrak{R} \otimes \bar{\mathfrak{R}}$), they are composed with the \otimes operator, since in our semiring-based formalism we do not have the concept of tokens (see Sec. 2).

5 The Secure NmSCCP Language

Given a soft constraint system as defined in Sec. 2 the syntax of NmSCCP agents is given in Fig. 3. Note that in this paper we present the syntax and extend the semantics (with rights) of the core subset of the actions defined in [6] (e.g., the *tell*, *ask*, *nask* and *retract* actions). In Fig. 3 P is the class of programs, F is the class of sequences of procedure declarations (or clauses), A is the class of agents, c ranges over constraints, X is a set of variables and Y is a tuple of variables.

One difference w.r.t. [6] is that the *tell* action has a new parameter (in addition to c), that is the $\bar{\mathfrak{R}}$ rights. When executing *tell*($c, \bar{\mathfrak{R}}$), it is not obviously possible to quantitatively impose more rights on c than c itself: therefore, the syntactic conditions on $\bar{\mathfrak{R}}$ when writing NmSCCP programs are that $\forall i. c \vdash \bar{\mathfrak{R}}_t[i], c \vdash \bar{\mathfrak{R}}_a[i], c \vdash \bar{\mathfrak{R}}_r[i]$.

A second difference w.r.t. [6] is represented by the new *execp* operation that can be used to execute a procedure $P(Y)$ in parallel, passing to the new agent

$P ::= F.A$
 $F ::= p(Y) :: A \mid F.F$
 $A ::= \text{secfail} \mid \text{success} \mid \text{tell}(c, \bar{\mathfrak{R}}) \succ A \mid \text{retract}(c) \succ A \mid \text{execp}(p(Y), \bar{\mathfrak{R}}) \mid E \mid A \parallel A \mid \exists x.A \mid p(Y)$
 $E ::= \text{ask}(c) \succ A \mid \text{nask}(c) \succ A \mid E + E$

Fig. 3. Syntax of the NmSCCP language with the new *execp* operation

$$\begin{array}{l}
\mathbf{R1} \frac{\sigma \neq \emptyset \quad \mathfrak{R}_i[i] \vdash c \quad \mathfrak{R}_i[i] = \mathfrak{R}_i[i] \ominus c \quad \text{check}(\sigma \otimes c) \rightarrow}{\langle \text{tell}^i(c, \bar{\mathfrak{R}}) \succ A, \sigma, \mathfrak{R} \rangle \rightarrow \langle A, \sigma \otimes c, \mathfrak{R} \otimes \bar{\mathfrak{R}} \rangle} \\
\mathbf{R2} \frac{\sigma = \emptyset \quad \mathfrak{R}_i[i] = \mathfrak{R}_i[i] \ominus c \quad \text{check}(\sigma \otimes c) \rightarrow}{\langle \text{tell}^i(c, \bar{\mathfrak{R}}) \succ A, \sigma, \mathfrak{R} \rangle \rightarrow \langle A, \sigma \otimes c, \mathfrak{R} \otimes \bar{\mathfrak{R}} \rangle} \\
\mathbf{R3} \frac{\mathfrak{R}_i[i] \vdash c \quad \mathfrak{R}'_i[i] = \mathfrak{R}_i[i] \ominus c \quad \sigma \vdash c \quad \sigma' = \sigma \ominus c \quad \text{check}(\sigma \ominus c) \rightarrow}{\langle \text{retract}^i(c) \succ A, \sigma, \mathfrak{R} \rangle \rightarrow \langle A, \sigma', \mathfrak{R}' \rangle} \\
\mathbf{R4} \frac{\mathfrak{R}_i[i] \vdash \bar{\mathfrak{R}}_i \quad \mathfrak{R}_a[i] \vdash \bar{\mathfrak{R}}_a \quad \mathfrak{R}_r[i] \vdash \bar{\mathfrak{R}}_r \quad p(Y) :: B \in F \quad \text{check}(\sigma) \rightarrow}{\langle \text{execp}^i(p(Y), \bar{\mathfrak{R}}) \succ A, \sigma, \mathfrak{R} \rangle \rightarrow \langle A \parallel B, \sigma, \mathfrak{R} \cup \bar{\mathfrak{R}} \rangle} \\
\mathbf{R5} \frac{\langle E_j, \sigma, \mathfrak{R} \rangle \rightarrow \langle A_j, \sigma', \mathfrak{R}' \rangle \quad j \in [1, n]}{\langle \sum_{i=1}^n E_i, \sigma, \mathfrak{R} \rangle \rightarrow \langle A_i, \sigma', \mathfrak{R}' \rangle} \\
\mathbf{R6} \frac{\mathfrak{R}_a[i] \vdash c \quad \sigma \vdash c \quad \text{check}(\sigma) \rightarrow}{\langle \text{ask}^i(c) \succ A, \sigma, \mathfrak{R} \rangle \rightarrow \langle A, \sigma, \mathfrak{R} \rangle} \\
\mathbf{R7} \frac{\mathfrak{R}_a[i] \vdash c \quad \sigma \not\vdash c \quad \text{check}(\sigma) \rightarrow}{\langle \text{nask}(c) \succ A, \sigma \rangle \rightarrow \langle A, \sigma \rangle} \\
\mathbf{R8} \frac{\langle A, \sigma, \mathfrak{R} \rangle \rightarrow \langle A', \sigma', \mathfrak{R}' \rangle}{\langle A \parallel B, \sigma, \mathfrak{R} \rangle \rightarrow \langle A' \parallel B, \sigma', \mathfrak{R}' \rangle} \\
\langle B \parallel A, \sigma, \mathfrak{R} \rangle \rightarrow \langle B \parallel A', \sigma', \mathfrak{R}' \rangle \\
\mathbf{R9} \frac{\langle A, \sigma, \mathfrak{R} \rangle \rightarrow \langle \text{success}, \sigma', \mathfrak{R}' \rangle}{\langle A \parallel B, \sigma, \mathfrak{R} \rangle \rightarrow \langle B, \sigma', \mathfrak{R}' \rangle} \\
\langle B \parallel A, \sigma, \mathfrak{R} \rangle \rightarrow \langle B, \sigma', \mathfrak{R}' \rangle \\
\mathbf{R10} \frac{\langle A[x/y], \sigma, \mathfrak{R} \rangle \rightarrow \langle B, \sigma', \mathfrak{R}' \rangle}{\langle \exists x.A, \sigma, \mathfrak{R} \rangle \rightarrow \langle B, \sigma', \mathfrak{R}' \rangle} \quad y \text{ fresh} \\
\mathbf{R11} \frac{\langle A, \sigma, \mathfrak{R} \rangle \rightarrow \langle B, \sigma', \mathfrak{R}' \rangle}{\langle p(Y), \sigma, \mathfrak{R} \rangle \rightarrow \langle B, \sigma', \mathfrak{R}' \rangle} \quad p(Y) :: A \in F
\end{array}$$

Fig. 4. The transition system for the secure NmSCCP

a subset of the owned $\bar{\mathfrak{R}}$ rights. We introduce this operation in order to model also the *eval* operation provided by the Linda language (see Sec. 3), where it is used to evaluate a further agent in parallel. Our *execp* is defined for the same aim, but in addition, in NmSCCP it is also used in order to pass different rights to the new agent (a real application of this operation is the fork of processes).

To give an operational semantics to our language we need to describe an appropriate transition system $\langle \Gamma, T, \succ \rangle$ where Γ is a set of possible configurations, $T \subseteq \Gamma$ is the set of *terminal* configurations and $\succ \subseteq \Gamma \times T$ is a binary relation between configurations. The set of configuration is $\Gamma = \{ \langle A, \sigma, \mathfrak{R} \rangle \}$ where $\sigma \in C$ and \mathfrak{R} is the matrix of rights, while the set of terminal configuration is instead $T = \{ \langle \text{success}, \sigma, \mathfrak{R} \rangle \}$. To remember even rights during the computation, in Def. 2 we extend the representation of a computation state w.r.t. [6].

Definition 2 (Computation States). *The state of a computation in NmSCCP is represented by a triple $\langle A, \sigma, \mathfrak{R} \rangle$, where A is the description of the agent to be executed, σ is the ongoing constraint store, and \mathfrak{R} is the current matrix of rights. \mathfrak{R} is initialized as $\forall i. \mathfrak{R}_i[i] = \emptyset, \mathfrak{R}_a[i] = \emptyset, \mathfrak{R}_r[i] = \emptyset$.*

In Fig. 4 we describe the transition system of the language. The actions of the agents are labeled with the identifier of the executing agent, in order to manage the changes and the security check related to the rights.

Tell: in **R1** and **R2** rules (see Fig. 4), the parameters of the action are represented by the constraint c , inserted by agent A_i , and by the new rights $\bar{\mathfrak{R}}$

representing *how much* of c can be respectively added ($\bar{\mathfrak{R}}_t$), asked ($\bar{\mathfrak{R}}_a$) or removed ($\bar{\mathfrak{R}}_r$) by the other agents. In this way, access rights are added to the matrix of rights \mathfrak{R} just as the new constraint c is added to the store σ .

If the *tell* represents the first step of the computation, that is if $\sigma = \emptyset$, we use rule **R2** where we do not check the *tell rights* for A_i . The reason is that, since $\mathfrak{R}_t = \emptyset$ at the beginning of the computation, it would not be ever possible to add a constraint since $\emptyset \not\vdash c$. Otherwise, if the store is not empty (i.e., **R1**), the *tell* operation can be executed if it also satisfies the *tell rights*, i.e., if $\mathfrak{R}_t[i] \vdash c$. If this holds, then the *tell rights* of A_i have to be updated according to the added c , i.e., $\mathfrak{R}'_t[i] = \mathfrak{R}_t[i] \ominus c$, in the sense that they are consumed. The actual matrix of rights \mathfrak{R} is updated with the passed new rights $\bar{\mathfrak{R}}$: therefore, the matrix of rights in the successive state corresponds to $\mathfrak{R}' = \bar{\mathfrak{R}} \otimes \mathfrak{R}$.

The new store $\sigma \otimes c$ must also satisfy the conditions of the specific \rightarrow transition, that is $check(\sigma')_{\rightarrow}$, according to the four cases shown in Fig. 2.

Retract: with **R3** we are able to “remove” a constraint c from the store σ , using the \ominus constraint division function as defined in Sec. 2. In this case we check if the *retract rights* of A_i are satisfied, that is if $\mathfrak{R}_r[i] \vdash c$. Moreover, we have also to check if c is entailed by the store, i.e., if $\sigma \vdash c$, in order to be able to effectively remove it. If these two conditions are satisfied, than the *retract* can be executed leading to the new store $\sigma' = \sigma \ominus c$ and the matrix of rights is updated as $\mathfrak{R}'_r[i] = \mathfrak{R}_r[i] \ominus c$: also in this case, as for the *tell* operation, A_i consumes part of its (*retract*) rights. The new store $\sigma' = \sigma \ominus c$ must also satisfy $check(\sigma')_{\rightarrow}$.

Execp: with **R4** we are able to create a new agent in parallel with the other already being executed. The “body” of the new agent is described by one of the procedures defined in the declaration section F , as defined in Fig. 3 in the precondition of the rule, $p(Y) :: B \in F$. The creating agent can pass to the son a part of his rights, and at most all of his rights. The precondition of this rule checks if the passed *tell/ask/retract* rights $\bar{\mathfrak{R}}$ (parameter of the *execp*) are entailed by the current rights owned by the creator agent. These rights are then passed to the son, but they are *not* revoked from the creator, i.e., subtracted with the operation $\mathfrak{R} \ominus \bar{\mathfrak{R}}$: the rights of the creator agent are not consumed. A new row is created in the matrix of rights, corresponding to the new running agent ($\cup \bar{\mathfrak{R}}$ in **R4**). The *execp* operation is equivalent to the Linda *eval* operation (see Sec. 3).

Nondeterminism: the composition operator $+$, represented in **R5** with \sum , can be applied to the E agents in Fig. 3, that is *ask* and/or *nask* agents. This rule nondeterministically chooses one of the agents whose guard succeeds, and clearly gives rise to global nondeterminism.

Ask: rule **R6** checks the presence of constraint c in the current store σ , that is if σ entails c (i.e., $\sigma \vdash c$). The other conditions are, in addition to $check(\sigma)_{\rightarrow}$, to check if $\mathfrak{R}_a[i] \vdash c$, that is to check if the *ask rights* $\mathfrak{R}_a[i]$ of the agent A_i performing the *ask* request entails c . Since the *ask* just represents a reading of the store, the σ store is not modified and the $\mathfrak{R}_a[i]$ rights are not consumed: as a result, the same c can be read several times with different successive *ask* operations.

Nask: rule **R7** is needed to infer the absence of a statement whenever it cannot be derived from the current state [6]. The rule is enabled when c is *not* entailed

by the store, i.e., $\sigma \not\vdash c$. Note that the rights checked in rule **R7** are just the same *ask rights* used by *ask* operations: the constraint added with a *tell* can be used for a positive or negative query in the store, and, therefore, the *ask rights* are enough to control both kinds of readings and hide part of such information.

Parallelism: a parallel agent (rules **R8** and **R9**) succeeds when both agents succeed. This operator is modeled in terms of *interleaving* (as in the classical CCP [16]): each time, the agent $A \parallel B$ can execute only one between the initial enabled actions of A and B (**R8**); a parallel agent succeed if all the composing agents succeed (**R9**).

Hidden variables: the semantics of the existential quantifier in **R10** is similar to that described in [16] by using the notion of *freshness* of the new variable added to the store. Hidden variables cannot be observed by an external observer.

Procedure calls: the semantics of the procedure call (**R11**) (i.e., the call to a different portion of code) has already been defined in [5]: the notion of diagonal constraints [16,5] is used to model parameter passing.

Note that the semantics given in Fig. 4 implements both the *Discretionary* and *Mandatory Access Control* principles (*DAC* and *MAC*) [13]. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing a permission to any other agent A_i : in our framework, this is enforced by \mathfrak{R} in rules **R1** and **R2**, that is the matrix of rights related to the constraint c added to the store by a *tell*. *DAC* is also enforced by the semantics of the *execp*, since rights are passed to the new agent. In addition, also the *MAC* model is enforced as well, since any operation by any subject on any object is tested against the set of authorization rules to determine if the operation is allowed or not: the semantics of the rules in Fig. 4 enforces these controls.

Moreover, the *execp* operation models the passing of rights during process *fork*-like operations in Operating Systems. The process executing an *execp* or a *tell* action can pass to other agents only the strictly necessary rights for accomplishing their tasks, according to the *principle of least privilege* [17].

Note that we consider that each agent knows the identifier of the other agents participating to the secure computation on the shared store, to be able to pass the rights to each of them. This is a general premise for a secure computation, as for example in Operating Systems Primitives (e.g., POSIX IPC) [7], where a shared memory is referenced in the compilation of both processes. Note that in this paper we consider a concurrent but not distributed execution of the agents, which run on the same processor and do not “migrate” to other units. In addition, also other works in literature define an identifier for each entity whose computation is controlled [10]. However, we can suppose that the identifiers of agents are instead names of (security) classes each agent belongs to. The rights for each class are then shared by all the included agents; in this way it is not necessary to know the exact number of the agents, if this represents a limitation.

Security Failures. If the computation does not succeed, clearly it useful to detect the operations that led to a security violation over the resources. To capture the security failures of agents during a computation, we add the transition rules in Tab. 5. *secfail* (see Fig. 3) is an agent indicating that a security violation

Table 2. The lattice of the colors/rights

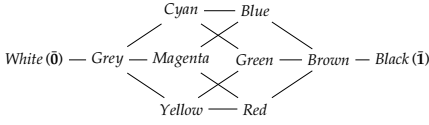


Table 3. The matrix of the rights after $tell(white, \mathfrak{R}_{A_C})$ by agent A_C

| | \mathfrak{R}_t (Tell) | \mathfrak{R}_a (Ask) | \mathfrak{R}_r (Retract) |
|-------|-------------------------|------------------------|----------------------------|
| A_C | <i>white</i> | <i>white</i> | <i>white</i> |

$$A_C :: \langle tell(white, \mathfrak{R}_{A_C}) \rightarrow_{white}^{black} except(P_{Red}, \mathfrak{R}_{A_R}) \rightarrow_{white}^{black} except(P_{Green}, \mathfrak{R}_{A_G}) \rightarrow_{white}^{black} except(P_{Blue}, \mathfrak{R}_{A_B}) \rightarrow_{grey}^{brown} nask(white) \rightarrow_{grey}^{brown} ask(yellow) \rightarrow_{yellow}^{brown} success, black \rangle$$

For the sake of readability, we do not detail the representation of the used constraints, but we just show the name of the color as a placeholder for the corresponding constraint. However, in the practice, we can use the cartesian product of three *Weighted* semirings, which is still a semiring [4], to represent the intensity of each primary color: for instance, $\langle x = 255, y = 0, z = 0 \rangle$ corresponds to preference *red* and $\langle x = 255, y = 255, z = 255 \rangle$ to *white*, according to the color theory. We only need three variables x, y and z to represent the problem. Therefore, we suppose $white = red \otimes green \otimes blu = \mathbf{0}$, since it contains all the colors, and $black = \mathbf{1}$ since it represents the absence of color (i.e., of information in the store). The lattice of the colors is represented in Tab. 2.

After the first *tell* of A_C , the global matrix of rights is shown in Tab. 3. A_C maintains the highest possible *tell/ask/retract rights* (i.e., *white*) for the color *white* it added to the store, since it is the principal controller of the screen resource. After the creation of the three sub-managers, the vector of rights $\mathfrak{R}_{A_R}, \mathfrak{R}_{A_G}$ and \mathfrak{R}_{A_B} lead to the matrix of rights depicted in Tab. 4. Note that A_C passes to the son agents only the necessary rights for accomplishing their tasks, according to the *principle of least privilege* [17]. For example, A_R has only the necessary *ask rights* to check if its managed color (i.e., *red*) is on the screen, but since it does not modify the store (the sequence *magenta, red* and *yellow* always contains *red*), both *tell* and *retract rights* are *black* (i.e., no rights).

The parallel computation then becomes $A_C \parallel A_R \parallel A_G \parallel A_B$, where the body of operations of A_R, A_G and A_B is respectively represented by P_{Red}, P_{Green} and P_{Blue} . Their description is shown in the following:

$$P_{Red} :: ask(red) \rightarrow_{white}^{black} success$$

$$P_{Green} :: retract(green) \rightarrow_{white}^{black} nask(magenta) \rightarrow_{white}^{black} tell(green, \mathfrak{R}_G) \rightarrow_{white}^{black} success$$

$$P_{Blue} :: nask(white) \rightarrow_{white}^{black} retract(blue) \rightarrow_{white}^{black} success$$

The vector of rights \mathfrak{R}_G , parameter of the $tell(green, \mathfrak{R}_G)$ in P_{Green} , is equal to $[black, black, black]$, since the A_G terminates after this operation and, for this reason, no additional rights are necessary. The matrix of rights at the end of the parallel computation is shown in Tab. 5. The *tell* and *retract rights* have been consumed by agents A_C and A_B . Therefore, it would not be possible for them to add or retract again any color from the screen.

Table 4. The matrix of the rights after the creation of A_R, A_G, A_B

| | \mathfrak{R}_t (Tell) | \mathfrak{R}_a (Ask) | \mathfrak{R}_r (Retract) |
|-------|-------------------------|------------------------|----------------------------|
| A_C | <i>black</i> | <i>white</i> | <i>black</i> |
| A_R | <i>black</i> | <i>red</i> | <i>black</i> |
| A_G | <i>green</i> | <i>magenta</i> | <i>green</i> |
| A_B | <i>black</i> | <i>white</i> | <i>blue</i> |

Table 5. The matrix of the rights at the end of the parallel computation

| | \mathfrak{R}_t (Tell) | \mathfrak{R}_a (Ask) | \mathfrak{R}_r (Retract) |
|-------|-------------------------|------------------------|----------------------------|
| A_C | <i>black</i> | <i>white</i> | <i>black</i> |
| A_R | <i>black</i> | <i>red</i> | <i>black</i> |
| A_G | <i>black</i> | <i>magenta</i> | <i>black</i> |
| A_B | <i>black</i> | <i>white</i> | <i>black</i> |

This parallel computation successfully terminates by showing all the desired three colors in sequence. But, for example, according to the matrix of rights shown in Tab. 4, the *secfail* agent could be achieved by A_G if it tries to remove color *yellow* instead of only *green* (see SF3 in Fig. 5), since *green* \neq *yellow* (see Tab. 2). Then, *secfail* would be propagated to the whole parallel computation, by using rule SF5 in Fig. 5.

Note that the presented example can be applied to more realistic scenarios with minor changes. For example, consider the three colors as Platinum, Gold and Silver traffic classes on QoS networks. QoS can then be monitored and enforced at an aggregate level (i.e., per-class).

7 Conclusions

In this paper, we have extended the NmSCCP language [6] in order to enhance with access rights the operations that can be performed on the store. Our model is based on the classical ACL security model: for each agent and operation (e.g., a constraint *retract* operation) we specify a quantity of information (i.e., a constraint) that that agent is allowed to operate on, that is, a quantity of information it is entitled to add, query, or remove. Therefore, both the resources and the access rights on them are represented with constraints. Since we adopt semiring-based soft constraints (see Sec. 2), the resources and the access rights are not represented as “crisp” pieces of information, as tuples in Linda [20] instead, or tokens in crisp constraint systems, as CCP [16]. The enhancement of CCP-like languages with security mechanisms is novel, even if possible extensions in this sense have been already anticipated in [12]. The presented model of rights is general, and it can be applied to any CCP language, wether “soft” or not.

As future work, we would like to prototype the presented soft security mechanisms with a constraint-based language, as, for example *Constraint Handling Rules* [8]. Moreover, we would like to use the lattice theoretic feature as a unify principle to combine SCCP with a Bell-LaPadula-like security system.

References

1. Bettini, L., Nicola, R.D.: A Java Middleware for Guaranteeing Privacy of Distributed Tuple Spaces. In: Guelfi, N., Astesiano, E., Reggio, G. (eds.) FIDJI 2002. LNCS, vol. 2604, pp. 175–184. Springer, Heidelberg (2003)

2. Bistarelli, S., Campli, P., Santini, F.: A secure coordination of agents with nonmonotonic soft concurrent constraint programming. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC 2012, pp. 1551–1553. ACM, New York (2012)
3. Bistarelli, S., Gadducci, F.: Enhancing constraints manipulation in semiring-based formalisms. In: ECAI 2006, 17th European Conference on Artificial Intelligence. Frontiers in Artificial Intelligence and Applications, vol. 141, pp. 63–67. IOS Press (2006)
4. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint satisfaction and optimization. *Journal of the ACM* 44(2), 201–236 (1997)
5. Bistarelli, S., Montanari, U., Rossi, F.: Soft concurrent constraint programming. *ACM Trans. Comput. Logic* 7(3), 563–589 (2006)
6. Bistarelli, S., Santini, F.: A nonmonotonic soft concurrent constraint language to model the negotiation process. *Fundam. Inform.* 111(3), 257–279 (2011)
7. Butenhof, D.R.: Programming with POSIX threads. Addison-Wesley Longman Publishing Co., Inc., Boston (1997)
8. Frühwirth, T.W.: Constraint Handling Rules. In: Podelski, A. (ed.) *Constraint Programming: Basics and Trends*. LNCS, vol. 910, pp. 90–107. Springer, Heidelberg (1995)
9. Gorla, D., Pugliese, R.: Resource Access and Mobility Control with Dynamic Privileges Acquisition. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) *ICALP 2003*. LNCS, vol. 2719, pp. 119–132. Springer, Heidelberg (2003)
10. Gorrieri, R., Lucchi, R., Zavattaro, G.: Supporting secure coordination in SecSpaces. *Fundam. Inform.* 73(4), 479–506 (2006)
11. Handorean, R., Roman, G.C.: Secure sharing of tuple spaces in ad hoc settings. *Electr. Notes Theor. Comput. Sci.* 85(3) (2003)
12. López, H.A., Palamidessi, C., Pérez, J.A., Rueda, C., Valencia, F.D.: A Declarative Framework for Security: Secure Concurrent Constraint Programming. In: Etalle, S., Truszczyński, M. (eds.) *ICLP 2006*. LNCS, vol. 4079, pp. 449–450. Springer, Heidelberg (2006)
13. McCollum, C.D., Messing, J.R., Notargiacomo, L.: Beyond the pale of MAC and DAC-defining new forms of access control. In: *IEEE Symposium on Security and Privacy*, pp. 190–200 (1990)
14. Nicola, R.D., Gorla, D., Pugliese, R.: On the expressive power of KLAIM-based calculi. *Theor. Comput. Sci.* 356(3), 387–421 (2006)
15. Omicini, A., Ricci, A., Viroli, M.: Agent coordination contexts for the formal specification and enactment of coordination and security policies. *Sci. Comput. Program.* 63(1), 88–107 (2006)
16. Saraswat, V., Rinard, M.: Concurrent constraint programming. In: *POPL 1990*, pp. 232–245. ACM (1990)
17. Schneider, F.B.: Least privilege and more. *IEEE Security & Privacy* 1(5), 55–59 (2003)
18. Tolone, W., Ahn, G.J., Pai, T., Hong, S.P.: Access control in collaborative systems. *ACM Comput. Surv.* 37, 29–41 (2005)
19. Udzir, N.I., Wood, A.M., Jacob, J.L.: Coordination with multcapabilities. *Sci. Comput. Program.* 64(2), 205–222 (2007)
20. Vitek, J., Bryce, C., Oriol, M.: Coordinating processes with secure spaces. *Sci. Comput. Program.* 46(1-2), 163–193 (2003)
21. Whitman, M.E., Mattord, H.J.: *Principles of Information Security*, 3rd edn. Course Technology Press, Boston (2007)

A Practical Approach for Closed Systems Formal Verification Using Event-B

Brett Bicknell¹, Jose Reis¹, Michael Butler², John Colley², and Colin Snook²

¹ Critical Software Technologies Ltd, 2 Venture Road, Southampton Science Park,
Southampton, SO16 7NP, United Kingdom
jreis@critical-software.co.uk,
<http://www.critical-software.co.uk>

² University of Southampton, Electronics and Computer Science, SO17 1BJ,
United Kingdom

Abstract. Assurance of high integrity systems based on closed systems is a challenge that becomes difficult to overcome when a classical testing approach is used; in particular the evidence generated from a classical testing approach may not meet the objectives of rigorous standards. This paper presents a new approach for the formal verification of closed systems, in particular commercial off the shelf (COTS) products. The approach brings together the formal language Event-B, mathematical proof theory and the Rodin toolset and provides the mechanism for creating abstract models of closed systems and to then verify these system properties against operational requirements. From an industrial perspective this approach represents a step change in the use and successful integration of closed systems; using formal methods to guarantee their integration and functionality. The outcome of the proof of concept will provide a solution that will increase the level of confidence on complex system of system solutions containing closed systems. Moreover, it will support the production of safety-cases by providing formal proofs of a system's correctness.

Keywords: closed systems, COTS, Event-B, Rodin, formal verification, virtualisation, VMWare.

1 Context

Closed Systems (CS) are becoming more common in the safety critical domain and in particular within the military domain. As the name connotes a closed system is a solution which is available as a black box where access to internal modules and design artifacts is not possible. In the context of this paper we will refer to a closed system as a software based system and of which can refer to either of the following contexts:

1. Systems developed by a third party that are commercially available to the public domain as readymade solutions; also known as COTS

2. Systems developed by a third party that are not commercially available but due to legal, export controls or other commercial reasons are not open to the system integrator

The justifications provided by system integrators for adopting a CS product as opposed to a bespoke solution are typically:

- Adoption of state of the art technology : The supplier of a CS has the budget to invest in R&D and innovative methods to provide state of the art technology. Furthermore these suppliers tend to invest in and continuously develop the product to stay competitive for longer
- Cost reduction : It is debatable whether a CS product offers a significant cost reduction or not, because the CS product may not be integrated seamlessly with the complete end system. If instead there are no significant issues with the integration of the CS, the end cost will be less than if the same solution was developed from scratch. Another key factor is that having the product on an open market pushes suppliers to offer competitive prices
- Reliability : In principle CS products have been tested thoroughly by the supplier, in that the solution is deemed reliable in particular contexts. The caveat is that the customer may have conditions that the solution has not been tested under and this becomes an issue, in particular when the CS is used in a safety critical system

The rollout of a CS product in a safety critical domain is not a straightforward activity as the assumptions made about the CS product may not hold when the product is integrated and tested with the rest of the system. It often happens that the CS product does not operate as expected in the specific environment or that some of the system requirements are not satisfied by the closed system. Furthermore the Verification and Validation (V&V) of Closed Systems is particularly difficult to perform because in most cases there is no design or requirements documentation available in the public domain, nor has the evidence of V&V activities performed by the supplier been made public. Determining the conditions and environment under which the CS product was validated is not simple. The non-existence of this data makes it more difficult to build a case for the reliability and safety of the end product.

Where all this might not be vital in a non-safety critical domain, it is particularly relevant in a safety critical domain where, for instance, the DO-178 standard requires evidence of V&V activities to meet specific objectives.

2 Existing Approaches

2.1 Integration Testing and Prototyping

The validation of CS based solutions typically starts as soon as the CS product is made available to the system integrator. First the product is taken through a prototyping stage with parts of the system being integrated with the CS products.

This stage validates some functional integration aspects; it does not necessarily address safety or reliability properties of the system as the focus is instead on validating the functional integration with the end system. The prototyping stage may lead to the identification of issues with the CS that require customizations, of which in some cases the supplier of the CS product can be open to fixing. This stage can also identify limitations which could lead to either rejecting the CS product or the system integrator having to identify workarounds.

Once the various elements of the solution are amenable to integration the system is formally built and only then can it go through integration testing. The approach used for testing is driven by the set of requirements including safety requirements, use cases, the test environment and last but not least the time available to perform the testing.

There are several risks associated with this approach:

- The identification of issues and the need for workarounds might come late in the lifecycle when there is no other option but to develop bespoke solutions. This impacts on the cost and schedule of the end system
- A limited test coverage as a result of the limited time available for testing and a limited test environment which does not allow for simulating the real in-service environment. This impacts on the confidence in the end solution and could result in a system deployed with untested states and an impact on the ability to generate a sound safety case
- As safety can be seen as an emergent property of the integrated system [3], the safety analysis, which could result in the need for significant re-design, occurs too late in the verification process

2.2 Fault Injection Testing

Fault injection testing is another approach that aims to address scenarios where there is insufficient evidence to demonstrate that the CS will not fail unacceptably or where the CS does not satisfy safety requirements. The primary objective of fault injection testing is to understand whether additional fault detection, isolation and recovery (FDIR) mechanisms are required or if the existing FDIR mechanisms are satisfactory. An example of a successful application of the method is described in [4].

The key risks associated with this method are:

- The APIs may not be sufficiently well documented resulting in additional effort to effectively customise the test environment and integrate it with the target system
- The results produced are constrained by the number of fault scenarios generated and as a consequence the evidence generated may be inadequate to demonstrate the robustness of the target system
- The fault injection tool may not provide all of the interfaces required to integrate with the equipment under test and as a consequence it may not be possible to inject faults in the target system

- Potential failures caused by unexpected component interactions in the integrated system are not addressed by this approach

There are a number of other approaches that can be used to facilitate the verification and validation of CS based solutions, such as using pre-existing evidence or reverse engineering. For further details see [5].

3 Solution Description

This paper outlines an approach that we have been applying to a real industrial project using the Event-B formal method. The system being analysed uses a number of complex static configurations and we were able to specify general requirements on these configurations in Event-B, and verify specific configurations using model checking and mathematical proofs. We also use refinement to map a high level model of the requirements to a lower level model that represents an abstraction of the CS product. Our experience to date suggests this approach encourages a deeper analysis of the system requirements and how these relate to the characteristics of the CS product than a less formal approach. The formal models and verification results also contribute to the safety case for the system.

The work on the case study is still in an intermediate stage, yet enough work has been completed to draw preliminary conclusions on the method and issues faced.

3.1 Tools Description

The case study uses the Event-B language, which is a formal language built upon set theory and mathematical proof, and is an evolution of the industrial-strength B method. This is in combination with the Rodin toolset, which is an Eclipse-based open source development environment for Event-B models [1]. Rodin includes proof obligation generation and automated proof tools along with a range of model-checking plug-ins. To date, Rodin has mostly been used in a research environment, yet using real industrial case studies.

The extensions to Rodin which are utilised in this case study are UML-B [6] - a UML-like graphical front end for Event-B which provides strong support for model refinement - and ProB [2], a plug-in which supports simulation and model-checking of both Event-B and UML-B models.

3.2 Case Study Overview

The system to be verified through the case study is a shared computer environment (SCE) which consists of hardware, virtualization middleware and applications. The hardware is composed of multiple servers, and the virtualization is provided by VMWare - a closed system - which enables multiple applications to be running on the same server and provides dynamic load-balancing and resource reallocation across the system.

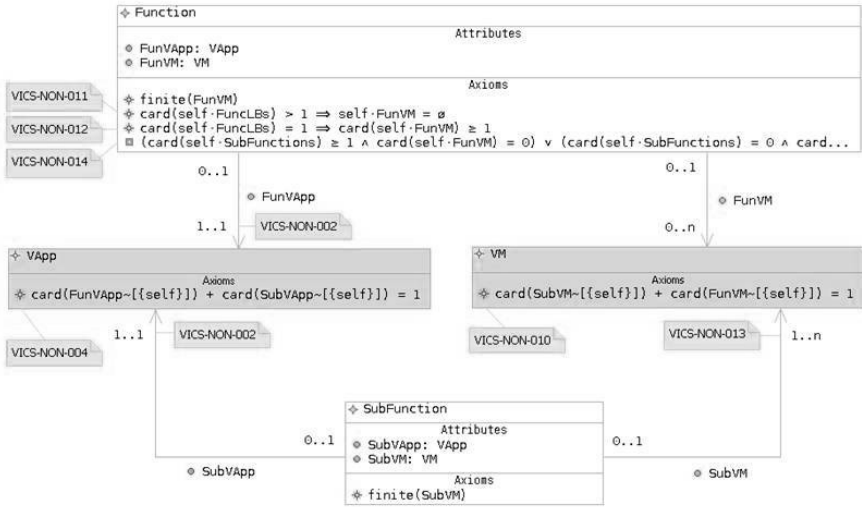


Fig. 1. Example of UML-B model Refinement

The abstract design for the system is broken down into logical servers, which represent groupings of applications that should be running on the same server. The hierarchy of the applications is introduced by the classification of each application to a function or sub-function; this allows for functions to span multiple logical servers if they have sub-functions running on each. In relation to VMWare, each function and sub-function corresponds to a Virtual Appliance (VApp) and can have multiple Virtual Machines (VMs) as depicted in the following figure:

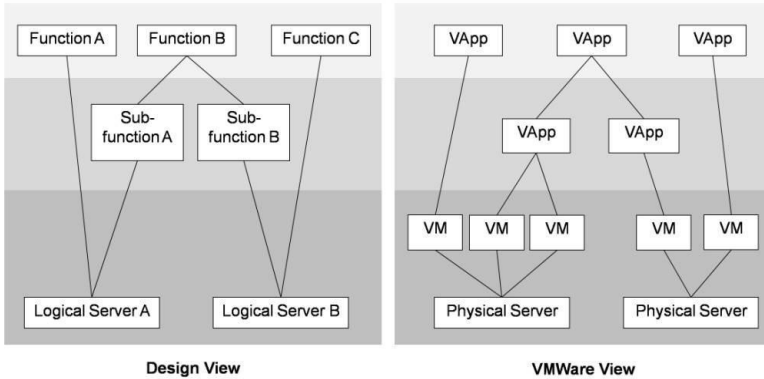


Fig. 2. Relationship between Design View and VMWare Architecture View

Each SCE operates according to a fixed number of static configurations mapping functions and sub-functions to logical servers. In this way it is possible to

define configurations for multiple SCEs, a single SCE (should one or more of the SCEs fail), or minimum safe configurations (so that the critical applications can continue running in the event of multiple server failures in the same SCE).

3.3 Modeling Strategy

The initial model developed has got four levels of refinement as shown in figure 3.

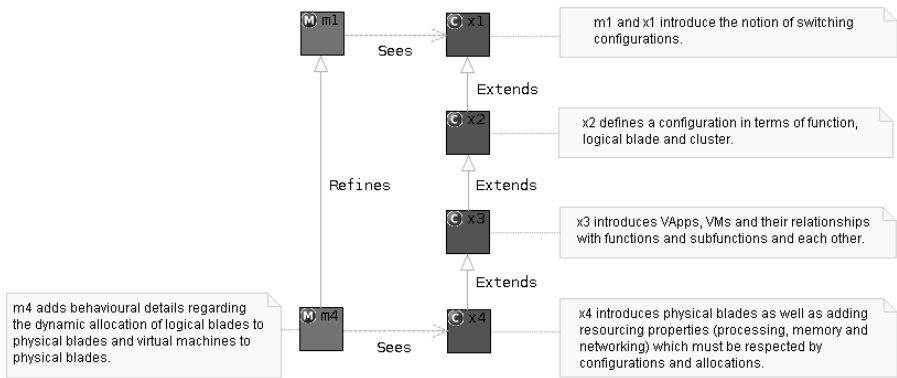


Fig. 3. Refinement Strategy

The properties of the system that can be verified formally are broken down into two phases; the static verification of each configuration, and the dynamic verification of the design requirements with the behavior of VMWare. The static phase of the verification involves proving that the configuration adheres to the requirements within the design - for instance, that a function assigned to a single logical server should not have any sub-functions (see figure 1) - but also that the memory, processing and network resources assigned to the sum of the functions and sub-functions running on each logical server do not exceed the limits on a single server (see figure 4).

VMWare can control the memory and processing resources; however, if the overall limits for the server are exceeded then it cannot guarantee that all of the applications will run correctly. Perhaps more critically, VMWare has no control over the network usage; so it is imperative that the network resource use is checked before each configuration is run. The model depicted in figure 4 introduces the concept of the memory, processing and network requirements of each VApp and the corresponding limits on each of the blades. The context includes these values as attributes as they are unchanging. The overall network limits are also introduced via two constants NetworkIngressLimit and NetworkEgressLimit.

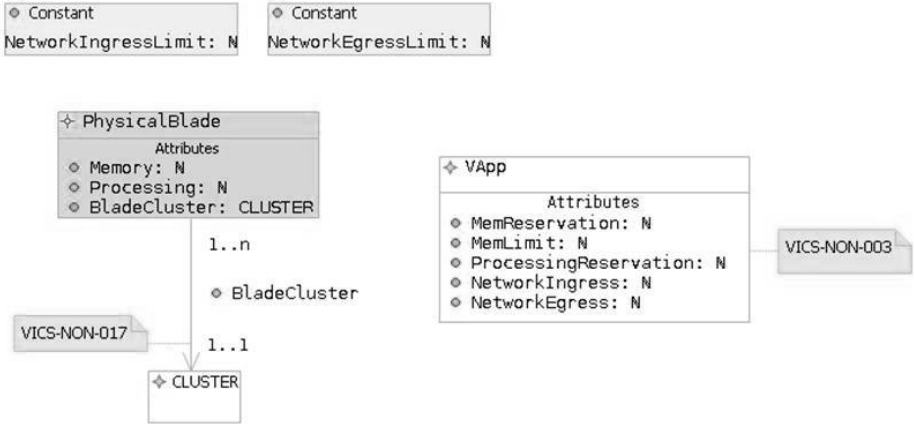


Fig. 4. UML-B model of physical resources and associated attributes

In each of the static cases the formal verification is achieved by modelling the requirements as axioms in the model and then inserting the mappings specific to the configuration into the model, examples of which describing the requirements on the memory are shown below:

AXIOMS

$$\begin{aligned}
 \text{Axiom10} : \forall LB \cdot LB \in \text{LogicalBlade} \Rightarrow \\
 (SUM(\text{MemReservation}[\text{FunVM}[\text{FuncLBs}^{-1}\{\{LB\}\}]]) + \\
 SUM(\text{MemReservation}[\text{SubVM}[\text{SubLB}^{-1}\{\{LB\}\}]]) < \text{BladeMemory}
 \end{aligned}$$

$$\text{Axiom11} : \forall v \cdot v \in \text{VM} \Rightarrow \text{MemLimit}(v) \leq \text{BladeMemory}$$

END

To verify the static design requirements we used ProB to check that the axioms are consistent, i.e. that ProB can find sets and constants that satisfy the axioms. By using ProBs constraint solving abilities, we discovered several modelling errors in the UML-B model when ProB could not find a solution that satisfied the axioms that represented the customer requirements for the relationships between functions, sub-functions. We then verified that an example configuration (also defined using axioms) was consistent in ProB and therefore satisfied the axiomatic constraints on configurations such as affinity rules and function allocations.

The advantage of using a model checker in this case is clear; if new configurations are designed, or existing configurations are changed slightly by adding new functions, then a static check can be simply and quickly performed on the configuration in an automated fashion without having to manually recalculate any of the resource limits or check that the mappings are valid.

To verify that an example configuration obeys the resource requirements we used the Rodin provers. The resource rules were expressed as a theorem which, if the configuration is valid, should follow from the axioms describing the resourcing required by the configuration. This involved time consuming interactive proof, even for a simple configuration. Axiom 10 presented above is one example where several hypotheses had to be added manually during the proof to "point" the provers in the correct direction due to the overall complexity of the expression. As a result we have proposed a new proof rule which will be added to the Rodin provers, thereby decreasing the time needed to prove future configurations.

The dynamic verification involves demonstrating that the combination of the system and VMWare behaves in such a way that the system design requirements are not invalidated, as these requirements are not inherent within VMWare. More specifically, there are certain affinity rules within the design, such as groupings of functions which have to be running on the same server, that were created to maximize the performance of those functions. These affinity rules help minimizing the overhead generated by the inter-server communication and ensure that the state of the functions within a group is consistent, i.e. in the event of a server hosting a group of functions failing then all those functions running in that server will have to be restarted and the system will not be left in a state where certain functions within the group are running whilst others are not as it would be the case if these affinity rules did not exist. These affinity rules could also be invalidated by VMWare Distributed Resource Scheduler (DRS) if it decides to reallocate applications due to an increased load on one or more of the servers. The potential issues are even more apparent when server failures are considered; in this case entire logical servers have to be reallocated or the configuration has to be changed, all by VMWare whilst retaining the design requirements and operational capacity of the system. The formal approach here, although not yet complete, will be to model the dynamic behavior of VMWare in Event-B, and verify this using the Rodin provers against the existing axioms in the static verification representing the design and resource requirements.

The key limitations to the approach are:

- The complete CS has to be modelled and thus assumptions have to be made about its behaviour. This is in fact what determines the scope of the modelling; for instance, it is assumed for the dynamic behaviour that the memory reallocation and low-level processes work reliably. However it is important to keep in mind that these are the type of issues that, as mentioned earlier, are often verified during the development of the CS.
- Another known limitation to this approach is inherent to the Event-B language and is due to the restricted set of requirements that are amenable to the language. Only a subset of the requirements could be modelled; it is not possible to model requirements concerning the performance of the system, for instance. This is not to say that this is not possible with other formal languages, however.

- A further limitation which has become apparent through the work is that the requirements presented are often in an unsuitable form for formal modeling, and consist of a mix of high and low levels. During the case study another, intermediate, requirements document had to be produced to fill the gap between the initial requirements and the formal model.

The modelling process required the participation of the customer fundamentally to clarify the requirements and design specified by the customer; at each step in the process models were produced, which raised questions, and after more information was gained the models were developed further. A benefit of using a formal method such as Event-B is that it encourages deep requirements analysis due to the completeness of requirements necessary for formal modeling; before and during the modelling missing and inconsistent requirements were found, most of which were corrected or included in the intermediate requirements document. A further benefit of the Event-B approach described is that it encourages the investigation, discovery and modelling of safety constraints much earlier in the verification process than traditional methods.

4 Preliminary Conclusions

The work on the case study is still in an intermediate stage, yet the following preliminary conclusions can be drawn:

- Static properties of VMs running in VMWare can be formalised using the Event-B method and verified using the Rodin toolset
- Event-B is a suitable method to verify properties which are not managed by VMWare but which are critical and have to be validated before running the virtualised applications
- The Event-B method offers the mechanisms to abstract specific details pertinent to the CS and focus on a sub-set of properties which are complex to verify using a classical testing approach. However models generated need to be verified to ensure they provide a correct representation of the system and for that to be done test cases have to be generated
- The proposed method does not exclude a level of testing because the formal models generated do not address all aspects of the system under test, for instance the method does not enable the verification of the system performance
- Event-B models of VMWare can be reused and expanded with more detail if required
- We found UML-B's class diagram notation very suitable for modelling this system. UML-B allowed us to construct a model very quickly and to explore different modelling choices, along with providing a mechanism to visualise the model and communicate it to others

Further work is going to be performed to formally verify the design requirements with the behavior of VMWare.

Acknowledgements. This document is an overview of UK MOD sponsored research and is released to inform projects that include safety-critical or safety-related software. The information contained in this document should not be interpreted as representing the views of the UK MOD, nor should it be assumed that it reflects any current or future UK MOD policy. The information cannot supersede any statutory or contractual requirements or liabilities and is offered without prejudice or commitment.

References

1. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in event-b. *International Journal on Software Tools for Technology Transfer (STTT)* 12(6), 447–466 (2010)
2. Leuschel, M., Butler, M.: Prob: an automated analysis toolset for the b method. *International Journal on Software Tools for Technology Transfer (STTT)* 10(2), 185–203 (2008)
3. Leveson, N.G.: *Engineering a safer world: Systems thinking applied to safety*. MIT Press, MA (2012)
4. Madeira, H., Some, R.R., Moreira, F., Costa, D., Rennels, D.: Experimental evaluation of a cots system for space applications. In: *Proceedings of the International Conference on Dependable Systems and Networks, DSN 2002*, pp. 325–330. IEEE (2002)
5. Menon, C., Hawkins, R., McDerimid, J.: Interim standard of best practice on software in the context of ds 00-56 issue 4. SSEI, University of York, Standard of Best Practice (1) (2009)
6. Snook, C., Butler, M.: Uml-b and event-b: an integration of languages and tools. In: *The IASTED International Conference on Software Engineering, SE 2008*, February 12-14 (2008)

Extensible Specifications for Automatic Re-use of Specifications and Proofs

Daniel Matichuk¹ and Toby Murray^{1,2}

¹ NICTA, Sydney, Australia*

{Daniel.Matichuk,Toby.Murray}@nicta.com.au

² School of Computer Science and Engineering, UNSW, Sydney, Australia

Abstract. One way to reduce the cost of formally verifying a large program is to perform proofs over a specification of its behaviour, which its implementation refines. However, interesting programs must often satisfy multiple properties. Ideally, each property should be proved against the most abstract specification for which it holds. This simplifies reasoning and increases the property’s robustness against later tweaks to the program’s implementation. We introduce *extensible specifications*, a lightweight technique for constructing a specification that can be instantiated and reasoned about at multiple levels of abstraction. This avoids having to write and maintain a different specification for each property being proved whilst still allowing properties to be proved at the highest levels of abstraction. Importantly, properties proved of an extensible specification hold automatically for all instantiations of it, avoiding unnecessary proof duplication. We explain how we applied this idea in the context of verifying confidentiality enforcement for the seL4 microkernel, saving us significant proof and code duplication.

1 Introduction

Formally verifying real software is expensive: proving a single property of a program’s implementation can require an order of magnitude more effort than to write the implementation [4,5]. To avoid expending this much effort on *every* property to be proved of an implementation, it is common to construct an abstract specification for the software and prove that the software’s implementation formally *refines* this specification. While this is expensive, subsequent reasoning can then be performed over the abstract specification. In practice, such proofs can require only a similar amount of effort as that to write the implementation [5].

The verification of the seL4 microkernel [4] provides a useful data-point, being to our knowledge the most extensive code-level verification ever performed of a general-purpose software artifact. A microkernel is a minimal operating system kernel; seL4 implements services such as threads, virtual address spaces, IPC,

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

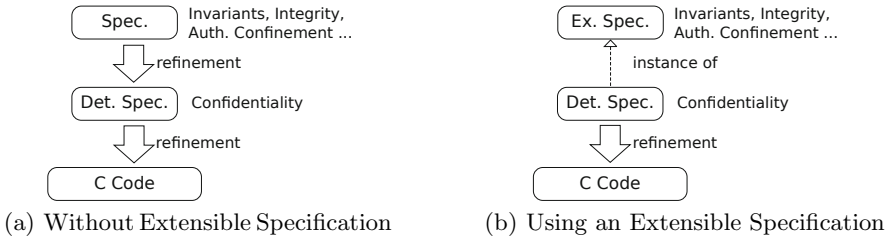


Fig. 1. Proving Confidentiality for seL4

and capability-based access control. An initial proof of refinement between the kernel’s C implementation and an abstract specification of its behaviour consumed about 25 person-years of effort [4]. This produced a proof on the order of 200,000 lines of Isabelle/HOL, including proving the basic kernel invariants for the abstract specification. However, subsequent proofs of security properties, namely integrity and authority confinement [9], have been carried out on the abstract specification, making use of these invariants, and then transferred to the C implementation via refinement. These proofs were completed in under 10 person months [9].

Reusable, general purpose software systems, such as operating system (OS) kernels, must often satisfy multiple properties. For instance, a secure OS kernel like seL4 should enforce not only integrity and authority confinement, but also confidentiality and availability. Unfortunately, writing a specification that captures all of these properties is tricky: one can reason about integrity in the presence of nondeterminism, but doing so with confidentiality under refinement is much harder because nondeterministic specifications tend to have insecure refinements [8].

seL4’s abstract specification, where integrity and authority confinement were proved, is nondeterministic. Under-specification of seL4’s scheduling behaviour is a major source of nondeterminism in this specification. The scheduling routine is maximally nondeterministic, saying only that the kernel can either (nondeterministically) schedule some runnable thread, or schedule the idle thread. Thus any sensible scheduling algorithm is a valid refinement of this scheduling specification. However, this includes malicious scheduling algorithms that might leak information via their scheduling decisions, perhaps by choosing the next thread to schedule by examining some secret information. For this reason, confidentiality cannot be proved about this specification, because it abstracts away from details that are relevant to confidentiality. We must therefore prove confidentiality of a less nondeterministic specification that, for instance, precisely specifies the kernel’s scheduling behaviour. We call this specification, the *deterministic* specification. This situation is depicted in [Figure 1\(a\)](#).

This might suggest that the initial abstract specification for seL4 was too nondeterministic. However, proving the kernel invariants, integrity and authority confinement at this level was the right thing to do. This is because, not only is a more abstract specification easier to reason about but, by proving these properties at a more abstract level, they become far more resilient to changes

in the kernel’s design and implementation, and we can still derive these properties for the deterministic specification by refinement. Specifically, having proved these properties about the nondeterministic specification, we can conclude that they hold for *all* possible refinements, which include all sensible scheduling implementations for instance. If we need to tweak an implementation detail of the scheduler, we need not fear that doing so will break these properties. The same does not necessarily follow if we have proved them only about the deterministic specification, which captures the precise scheduling behaviour.

This suggests that properties should be proved at the most abstract level at which they still hold. However, in the worst case, each property would require its own specification, as well as associated proofs of refinement between them. Any changes to the most abstract specification, such as an API evolution, must be reflected in all other specifications, and all proofs updated. This is expensive when APIs continually evolve and specifications duplicate much of each other’s code, as happens with the nondeterministic and deterministic seL4 specifications.

In this short paper, we present *extensible specifications*, a technique for constructing specifications that avoids these problems while still allowing properties to be proved at the highest levels of abstraction. This technique is designed primarily for very large mechanical proof efforts, with large bodies of existing proofs and specifications, where duplicating existing artifacts and performing and maintaining unnecessary proofs is undesirable. These ideas have been developed and formalised within the proof assistant Isabelle/HOL; however, they should be applicable to any other proof assistant for higher order logic.

2 Extensible Specifications

The seL4 specifications are formalised as nondeterministic state monads [2] and we explain extensible specifications with reference to this formalism.

An imperative program may be specified as a nondeterministic state monad by defining a *state type* that is a record containing a field for each global variable in the program and each relevant piece of global state (such as the state of external devices with which the program interacts). The program is then a function that takes one of these records as its input and yields an updated record and a return value as its output. Nondeterministic computations yield a set of such outputs. Traditional “do-notation”, as supported by Haskell for instance, is used to phrase monadic specifications in an imperative style.

A Running Example. We use a toy example program to motivate and explain extensible specifications throughout this section. This program contains two functions of interest, called `alloc` and `dealloc` whose signatures are:

```
int alloc(void);    void dealloc(int i);
```

`alloc` takes no arguments and allocates a new resource, returning an integer ID to the allocated resource. `dealloc` takes an ID as its argument that represents an allocated resource, and deallocates the resource if it is currently allocated, and does nothing otherwise. For simplicity, `alloc` is allowed to fail if there are no

| | |
|--|---|
| <pre> alloc-abs ≡ do ids ← gets ids; assert (ids ≠ ∅); i ← select ids; modify (ids-update (λf. f - {i})); return i od </pre> | <pre> dealloc-abs i ≡ do ids ← gets ids; when (i ∉ ids) (modify (ids-update (λf. f ∪ {i}))) od </pre> |
|--|---|

Fig. 2. An example abstract specification

resources to allocate. [Figure 2](#) depicts abstract specifications of these functions. The state type for these specifications is a record that contains a single field, `ids`, which holds a set of integers representing the IDs of currently free resources.

Given a field name `x`, `gets x` reads from the state record the value stored in field `x`, while `modify (x-update func)` updates the `x`-field of the state record with the result of running the function `func` on the current value stored in that field. Given a set `S`, `select S` nondeterministically selects a value from `S`. Hence, `alloc-abs` first reads the set of free IDs and asserts that it is not empty. It then nondeterministically selects from this set the next ID to allocate, before updating the set of free IDs in the state record by removing the chosen ID from it. Finally `alloc-abs` returns the chosen ID to its caller.

It is straightforward to show certain correctness conditions about these functions. For instance, we can prove that, whenever it is successful, `alloc-abs` returns the unique resource ID `i` that is now allocated but was originally free. This statement may be written in a monadic Hoare logic variant [\[2\]](#) as:

$$\{\lambda s. \text{ids } s = X\} \text{ alloc-abs } \{\lambda i s'. i \notin \text{ids } s' \wedge \text{ids } s' \cup \{i\} = X\} \quad (1)$$

This statement is read as follows: if, before `alloc-abs` executes from some pre-state `s`, `ids s = X`, then whenever it terminates, returning a result `i` in some post-state `s'`, `i ∉ ids s' ∧ ids s' ∪ {i} = X`.

`alloc-abs` completely abstracts away from the order in which resources are allocated. Reasoning about this order requires a more concrete specification of `alloc`'s behaviour. Suppose `alloc` is implemented by having it maintain a list of unused resource IDs, from which it selects the first item (and fails when this list is empty). [Figure 3](#) depicts a hypothetical concrete specification of this behaviour. This specification operates on a state record that extends the original by adding a new field `ids-list` that contains a list of currently free resource IDs. The original set `ids` is retained to allow existing properties like [\(1\)](#) to be phrased over this new specification. While not really necessary in this example, this is vital for larger specifications with a massive body of existing proof, such as `seL4`.

Extensible Specifications. Notice that much of the code in [Figure 3](#) is duplicated from [Figure 2](#). Also, proving an analogous result to [\(1\)](#) for `alloc-conc` requires re-proving it directly or concluding it from a proof of refinement between `alloc-abs` and `alloc-conc`. For large specifications, either approach requires significant effort.

By defining a single *extensible specification* for `alloc` that subsumes both `alloc-abs` and `alloc-conc` (and doing the same for `dealloc`), we can avoid this

| | |
|--|---|
| <pre> alloc-conc \equiv do ids-list \leftarrow gets ids-list; assert (ids-list \neq []); i \leftarrow return (hd ids-list); modify (ids-update ($\lambda f. f - \{i\}$)); modify (ids-list-update tl); return i od </pre> | <pre> dealloc-conc $i \equiv$ do ids \leftarrow gets ids; when ($i \notin$ ids) (do modify (ids-update ($\lambda f. f \cup \{i\}$)); modify (ids-list-update ($\lambda l. i.l$))) od </pre> |
|--|---|

Fig. 3. A hypothetical concrete specification

| | |
|--|--|
| <pre> alloc-ext select-ext update-ext \equiv do ids \leftarrow gets ids; more \leftarrow gets more; next-ids \leftarrow return (select-ext ids more); g-ids \leftarrow return (guard-set ids next-ids); assert (g-ids \neq \emptyset); i \leftarrow select g-ids; modify (ids-update ($\lambda ids. ids - \{i\}$)); modify (more-update (update-ext i)); return i od </pre> | <pre> guard-set ids-set next-ids-set \equiv if next-ids-set \subseteq ids-set \wedge next-ids-set \neq \emptyset then next-ids-set else ids-set dealloc-ext i update-ext \equiv do ids \leftarrow gets ids; when ($i \notin$ ids) (do modify (ids-update ($\lambda f. f \cup \{i\}$)); modify (more-update (update-ext i))) od </pre> |
|--|--|

Fig. 4. An example extensible specification

effort. A specification is made extensible by augmenting its state type with an extra *extended state* component of arbitrary type, and inserting carefully placed *extended computation* points in its code, which are placeholders for code that operates on the extended state. Extended computations are placed (1) where the extended state is read to resolve nondeterminism that abstracts away implementation choices, and (2) where extended state needs to be updated to ensure consistency with the program’s implementation. By instantiating the extended state with a concrete state type and the extended computations with specification code that operates over this state, one produces an *instance* of the extensible specification.

Figure 4 depicts extensible specifications for `alloc` and `dealloc`. As we will show, each may be instantiated to behave like the abstract and concrete specifications above, avoiding the duplication between them.

Importantly, we can also prove properties about these extensible specifications. Such properties hold for *all* instances of these specifications. This avoids proof duplication and/or having to maintain refinement proofs between specifications like `alloc-abs` and `alloc-conc`. We can easily rephrase **(1)** to be over `alloc-ext`:

$$\{\lambda s. \text{ids } s = X\} \text{ alloc-ext select-ext update-ext } \{\lambda i s'. i \notin \text{ids } s' \wedge \text{ids } s' \cup \{i\} = X\}$$

Due to the similarity between `alloc-ext` and `alloc-abs`, the proof of **(1)** is easily adapted to `alloc-ext`. This is true for any property of the original specification.

Because it holds for all instances of `alloc-ext`, the above Hoare triple automatically applies to the instantiations shown below for `alloc-abs` and `alloc-conc`.

`alloc-ext` and `dealloc-ext` operate over a state type that extends the original state type with an extra field, `more`, of arbitrary type. They are parametrised by subroutines that perform extended computation (*select-ext* and *update-ext*). While we have included them in this paper for ease of presentation, passing the extended computations as parameters can be avoided by defining them as abstract operations on a *type-class* that the extended state implements. This approach should also work for other higher order logic proof assistants, like Coq.

We obtain `alloc-ext` from `alloc-abs` by adding two blocks of extended computation. The first reads the extended state and uses it to help resolve the non-deterministic selection of the next ID to allocate, while the second updates the extended state following this selection.

`alloc-ext` is carefully constructed so that it behaves like `alloc-abs` with respect to the non-extended state. `alloc-ext` calls the extended computation *select-ext* to obtain the set of IDs from which to subsequently select the ID to allocate. Importantly, `alloc-ext` then makes use of the function `guard-set` whose purpose is to force `alloc-ext` to behave like `alloc-abs` no matter what *select-ext* did: `guard-set` ensures that the subsequent `assert` fails only when `ids` is empty and the subsequent `select` always chooses an ID from `ids`. `guard-set` does this by checking that the set chosen by *select-ext* is a non-empty subset of `ids` and replacing it by `ids` if it is not. By building `alloc-ext` this way, we ensure that any property unrelated to the extended state that `alloc-abs` satisfies also holds for all instantiations of `alloc-ext`.

The original specifications of [Figure 2](#) can be recovered by instantiating the extended state to be of type *unit* (the type with one element), and the extended computations to be no-ops. For `alloc-ext`, we have *select-ext* return the entire set `ids`, which ensures that the subsequent `select` behaves exactly like the nondeterministic `select` in `alloc-abs`. Finally, we have *update-ext* leave the extended state unchanged. Hence `alloc-abs` = `alloc-ext` ($\lambda ids\ more.\ ids$) ($\lambda i\ more.\ more$) and `dealloc-abs` i = `dealloc-ext` i ($\lambda i\ more.\ more$).

We can also instantiate `alloc-ext` to behave like `alloc-conc` from [Figure 3](#) to reason about the concrete behaviour of `alloc`. We instantiate the extended state to include a list of currently unallocated resource IDs. We define the functions `ids-list-ext`, which reads this list from the `more` field, and `ids-list-update-ext`, which updates it; we omit these definitions for brevity. We then have *select-ext* return the singleton set containing the head of this same list, and have *update-ext* modify this list by replacing it with its tail (i.e. by removing its first item).

`alloc-ext` will behave deterministically, by performing the `select` from the singleton set given by *select-ext*, so long as this set is contained in `ids`, to prevent `guard-set` causing selection from the entirety of `ids`. `alloc-ext` behaves the same as `alloc-conc` in this case so long as `ids-list` is empty if and only if `ids` is. The invariant `valid-list` ensures this, and is trivial to prove of the deterministic instantiation.

$$\text{valid-list } s \equiv \text{distinct } (\text{ids-list } s) \wedge \text{set } (\text{ids-list } s) = \text{ids } s$$

It states that each ID in `ids-list` is distinct, and each ID in `ids-list` is in `ids` and vice-versa. Under this invariant, the instantiated extensible specification exhibits the

hypothetical concrete behaviour precisely. Specifically, $\text{valid-list } s \longrightarrow \text{alloc-conc } s = \text{alloc-ext } (\lambda \text{ids } \text{more. } \{\text{hd } (\text{ids-list-ext } \text{more})\}) (\lambda \cdot \text{ids-list-update-ext } \text{tl}) s$ and $\text{dealloc-conc } i = \text{dealloc-ext } i (\lambda i. \text{ids-list-update-ext } (\lambda l. i \cdot l))$.

These kinds of invariants, which assert consistency between the instantiated extended state and the non-extended state, are commonly required for reasoning that a concrete instantiation of an extensible specification behaves correctly with respect to the extended state. They are also the same kind of invariants required to prove refinement between alloc-abs and alloc-conc , for instance; although extensible specifications avoid the need for this additional refinement proof.

3 Proving Confidentiality for seL4

We now explain how we applied extensible specifications to assist proving confidentiality of the seL4 microkernel. Our requirement for proving confidentiality was having a deterministic specification, with proofs of the thousands of lemmas that establish the correctness of the individual kernel functions, and proofs of the kernel invariants, integrity and authority confinement on this specification. These results have already been proven for seL4’s nondeterministic abstract specification. Extensible specifications allow us to obtain these results for the deterministic specification as well without unnecessary effort.

The abstract seL4 specification is approximately 5,500 lines of Isabelle/HOL, and the proofs of integrity and authority confinement, and the kernel correctness lemmas and invariants, comprise around 65,000 lines of Isabelle/HOL. However, nondeterminism is used to abstract away from implementation details in only three main places: the precise order in which hardware address space identifiers (ASIDs) are allocated, the order in which capabilities are recursively deleted during a *revoke* system call, and to abstract away the scheduling algorithm, as explained earlier. If we were to take the approach depicted in [Figure 1\(a\)](#) and manually define a deterministic abstract specification for seL4, it would duplicate around 98% of the abstract specification. We would also have to prove refinement between this new specification and the original, and maintain this proof going forward. This refinement theorem would have to be repeatedly applied to prove the thousands of correctness lemmas, as well as the kernel invariants, integrity and authority confinement for the deterministic specification.

We avoided these problems by altering about 2% of the abstract specification to make it extensible, as depicted in [Figure 1\(b\)](#) where the dashed arrow indicates that the deterministic specification is an instance of the extensible one. We repeated the process shown in [Section 2](#) for each nondeterministic function in the specification. [Section 2](#)’s example is a simplification of the cases for hardware ASID allocation and capability revocation: each of these involves replacing a nondeterministic selection with a deterministic one, based on some extra state that the deterministic specification must track. We are currently designing a confidentiality-preserving scheduler for seL4. Once complete, we will modify the current extensible specification to allow scheduling decisions to be

implemented by extended computations. Rephrasing the invariants and correctness lemmas, authority confinement and integrity properties and adapting their proofs to the extensible specification altered only $\sim 1,000$ lines of Isabelle/HOL ($\sim 1.5\%$). These results then hold for the original specification and the deterministic one without further effort.

By making the seL4 abstract specification extensible, we have avoided duplicating tens of thousands of lines of proof and specification code, and performing unnecessary refinement proofs. This would have required significant effort, and made the resulting artifacts a nightmare to maintain as seL4's API evolves.

4 Related Work

The basic ideas of extensible specifications are certainly not new. The extended state, and its abstract extended operations, which parametrise `alloc-ext` for instance, define the interface of an abstract data type [7] that instances of the extensible specification implement. Extensible specifications also resemble a lightweight form of Aspect-Oriented Programming [3], where our concrete extended computations resemble *advice*s and the sites at which they are placed resemble *join points*. When making a specification extensible, the appropriate join points are sites where extended state needs to be read to resolve nondeterminism, and sites where extended state needs to be updated.

While presented here in the context of state monads, extensible specifications also resemble mechanisms for specification and proof re-use within the B method (e.g. [1]) and Event-B (e.g. [10]). With these sorts of methods, a generic pattern can also carry assumptions that instances of it must meet in order to inherit results proved for the pattern. We can do likewise for extensible specifications by attaching assumptions to the type-class of the extended state. Monadic extensible specifications are arguably cleaner and simpler than these other methods, largely because they inherit the elegance and power of higher order logic for abstracting over extended computation.

Sophisticated verification systems that support automated stepwise refinement also offer similar benefits to extensible specifications. For instance, Chalice [6] allows the differences between two specifications to be encoded using *skeleton* syntax, and refinement between them automatically proved via SMT. This avoids duplicating code between specifications, and having an explicit refinement proof. We expect SMT could also be used to prove properties for the concrete specification already shown of the more abstract one. Unlike extensible specifications where proof re-use comes for free, here it relies on SMT solving.

5 Conclusion

We have presented extensible specifications, a lightweight technique for constructing specifications that can be instantiated and reasoned about at multiple levels of abstraction. By using extensible specifications one avoids having to write and maintain a different specification for each property being proved of a program,

whilst still allowing properties to be proved at the highest levels of abstraction. Properties proved of an extensible specification hold automatically for all instantiations of it, avoiding unnecessary proof duplication. This technique has been vital in assisting the ongoing proof of confidentiality for the seL4 microkernel, where it saved duplicating tens of thousands of lines of proof and specification code, and for maintaining these artifacts as the kernel has continued to evolve during this proof effort. Our experience applying extensible specifications to seL4 suggests that they are practically applicable and scale to real-world verification efforts.

Acknowledgements. Thanks to Gerwin Klein, David Greenaway and Mark Staples for valuable feedback on earlier drafts of this paper.

References

1. Blazy, S., Gervais, F., Laleau, R.: Reuse of Specification Patterns with the B Method. In: Bert, D., Bowen, J.P., King, S. (eds.) ZB 2003. LNCS, vol. 2651, pp. 40–57. Springer, Heidelberg (2003)
2. Cock, D., Klein, G., Sewell, T.: Secure Microkernels, State Monads and Scalable Refinement. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 167–182. Springer, Heidelberg (2008)
3. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
4. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: 22nd SOSP, Big Sky, MT, USA, pp. 207–220. ACM (October 2009)
5. Klein, G., Murray, T., Gammie, P., Sewell, T., Winwood, S.: Provable security: How feasible is it? In: 13th HotOS, Napa, CA, USA, pp. 28–32. USENIX (May 2011)
6. Leino, K.R.M., Yessenov, K.: Stepwise refinement of heap-manipulating code in Chalice (2011) Unpublished manuscript, <http://research.microsoft.com/en-us/um/people/leino/papers/krm1211.pdf>
7. Liskov, B., Zilles, S.: Programming with abstract data types. SIGPLAN Not. 9(4), 50–59 (1974)
8. Murray, T., Lowe, G.: On refinement-closed security properties and nondeterministic compositions. In: 8th AVoCS, Glasgow, UK. ENTCS, vol. 250, pp. 49–68 (2009)
9. Sewell, T., Winwood, S., Gammie, P., Murray, T., Andronick, J., Klein, G.: seL4 Enforces Integrity. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 325–340. Springer, Heidelberg (2011)
10. Silva, R., Butler, M.: Supporting Reuse of Event-B Developments through Generic Instantiation. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 466–484. Springer, Heidelberg (2009)

Implementing Tactics of Refinement in CRefine

Madiel Conserva Filho and Marcel Vinicius Medeiros Oliveira

Universidade Federal do Rio Grande do Norte – Brazil
madielfilho@gmail.com, marcel@dimap.ufrn.br

Abstract. *Circus* is a formal language that combines Z and CSP, providing support for specification of both data and behavioural aspects of concurrent systems. Furthermore, *Circus* has an associated refinement calculus, which can be used to develop software in a precise and step-wise fashion. Each step is justified by the application of a refinement law (possibly with the discharge of proof obligations). Sometimes, the same laws can be applied in the same manner in different developments or even in different parts of a single development. A strategy to optimize this calculus is to formalise this application as a refinement tactic, which can then be used as a single transformation rule. CRefine was developed to automate the *Circus* refinement calculus. However, before the work presented here, it did not provide support for refinement tactics. In this paper, we present an extension to CRefine: a new module that automates the process of defining and applying refinement tactics formalised in the tactic language ArcAngelC. Finally, we illustrate the usefulness of the extension in the development of an industrial case study.

Keywords: *Circus*, CRefine, ArcAngelC, refinement tactics.

1 Introduction

Circus [4] is a formal language that can be used to specify concurrent and reactive systems. It is a combination of Z [15] and CSP [14]: the former can be used to model systems with complex data structures and the latter is specific to concurrent systems and defines the concurrent behaviour of the system. Besides the specification of data and behavioural aspects of concurrent systems, *Circus* has an associated refinement calculus [10]. This calculus consists of repeated application of refinement laws to an initial abstract specification to produce a concrete specification. Using a refinement calculus, programs can be developed correctly in a stepwise fashion. Each step is an application of a refinement law, which is usually valid under certain conditions that need to be proved.

Sometimes, during the development using refinement calculus, the same laws are applied in the same manner in various developments or even in different parts of a single development. A strategy to optimize this calculus is to formalise these applications as refinement tactics, which can then be used as single transformation rules. Using this approach, the refinement calculus becomes more agile, reducing time and effort.

The manual development using refinement calculus is a hard and error-prone task because it encompasses many refinement laws in mostly long and repetitive

developments. CRefine [12]¹ was developed to support the application of the *Circus* refinement calculus. It automatically manages the development and its proof obligations, most of which are automatically proved. However, the previous version of CRefine did not support the definition and use of refinement tactics.

This paper presents an extension to CRefine, which adds the support for the definition and application of refinement tactics to CRefine. This extension constitutes a useful addition that can be used while modelling systems in *Circus*. Using the new module, users can define and use tactics that considerably optimise the *Circus* development process. The tactic language supported is ArcAngelC [13], a refinement tactic language for *Circus* programs that is similar to the tactic language for sequential programs ArcAngel [11] that is equally supported by Refine [11]. Both languages are based on the general language Angel [9].

The main addition to ArcAngelC comparing to ArcAngel is the possibility of defining tactics that can be applied to different classes of elements within a *Circus* specification: actions, processes and programs. ArcAngel tactics can only be applied to sequential programs. ArcAngelC has a formal semantics, which is based on ArcAngel's semantics, but adds some generality to support all extra *Circus* structural combinators. Using ArcAngelC, we can formalize basic tactics, as law applications, and tacticals, which are tactics combinators. Most of the structural combinators, which allow tactic application to specific parts of a program, are inherited from ArcAngel. However, structural combinators for *Circus* specific constructs, like those from CSP, are also part of ArcAngelC.

The remaining of this paper is structured as follows. The next section provides an overview of *Circus*. In Section 2, we briefly present CRefine and its features. The *Circus* tactic language, ArcAngelC, is described in Section 3. The new module of CRefine that provide support for refinement tactics is presented in Section 4. Section 5 presents the use of CRefine and its extension in an industrial case study. Finally, in Section 6 we make our final considerations and discuss future work.

2 CRefine

CRefine [12] provides support for the *Circus* refinement calculus. It plays an important role in the development process by providing an automatic management of the overall development and tools for documenting it. The automatic management includes a filter that displays to the user only those refinement laws that are applicable to the term selected, the application of the refinement laws that transforms the term selected possibly generating proof obligations (hereafter called POs), most of which are automatically proved.

CRefine GUI is composed of a main menu and three main frames: refinement, code and proof obligations. The first frame displays all the refinement steps of the development, including the result of the law applications. The current result of the refinement process is displayed in a second frame. Finally, the last frame lists all POs that have been generated in the development and marks them as

¹ CRefine can be downloaded from <http://www.cs.york.ac.uk/circus> - this distribution also contains the tactics module presented here along with examples.

valid, invalid, or unknown. Valid POs are those that have been automatically proved correct by CRefine’s prover module, which besides reasoning on syntactic restrictions also integrates with external SAT solvers like VeriT [2] using the ZB2SMT package [8]. The invalid POs are those that have been proved incorrect by the same module; they indicate an error in the development. Finally, POs marked as unknown are those that could not be evaluated by CRefine; they need to be manually verified by the user.

Using CRefine the user starts with a L^AT_EX document that follows the CZT syntax for *Circus*. This document contains the abstract specification to which we repeatedly apply refinement laws using CRefine. The law application consists of selecting the term in the development frame, and choosing an applicable law in the pop-up menu list. Some law applications requires arguments that are input by the user. Afterwards, the law is automatically applied: CRefine updates all frames based on the result of the application.

3 ArcAngelC

In ArcAngelC a tactic is declared as **Tactic** $N(args)$ *tactic* **end**. The declaration is composed of a tactic name N , the tactic’s arguments *args*, if any, and a tactic program *tactic*. The declaration can also include two optional clauses, **proof obligations**, which lists the POs generated by the tactic application, and **generates**, that documents the program generated. These clauses are used for documentation purposes only and do not have any consequence in its application.

ArcAngelC has five basic tactics. The tactic **law** $L(args)$ p applies a refinement law L using arguments *args*. A similar construct is **tactic** $T(args)$, which applies a given tactic T using arguments *args*. The tactic **skip** always succeeds and the tactic **fail** always fails. The inclusion of recursion in the language (explained later in this section) introduces an extra possibility for the eventual outcome of a tactic. As well as succeeding or failing to apply, it may fail to terminate and run indefinitely. Whilst such a tactic will not in general be useful when writing tactic programs, it is helpful to be able to reason about it. We follow Dijkstra [5] and call the non-terminating tactic **abort**.

The tactic **applies to** p **do** t is given a meta-program (or program pattern) p that characterises the programs to which the tactic t is applicable; the meta-variables used in p can then be used in t . For example, $A[ns_1 \mid cs \mid ns_2]Skip$ is a meta-program that characterises those parallel compositions whose right-hand action is *Skip*; here, A , ns_1 , cs and ns_2 are the meta-variables.

The tactical $t_1 ; t_2$ applies t_1 , and then applies t_2 to the outcome of the application of t_1 . If either t_1 or t_2 fails, so does the whole tactic. When it succeeds, the proof obligations generated are those resulting from the application of t_1 and t_2 . Tactics can also be combined in alternation: $t_1 \mid t_2$. First t_1 is applied. If that succeeds, then the composite tactic succeeds; otherwise t_2 is applied. If then the application of t_2 succeeds then the composite tactic succeeds; otherwise the composite tactic fails. If one of the tactics aborts, the whole tactic aborts.

By way of illustration, the tactic below uses alternatives. It promotes local variables declared in a main action to state components. This is the result of an application of either Law 1 (`prom-var-state`) or Law 2 (`prom-var-state-2`) depending on whether the process has state or not.

```
Tactic promoteVars()  $\hat{=}$  law prom-var-state() | law prom-var-state-2() end
```

The standard form of angelic choice is commutative. *ArcAngelC*'s choice, however, as denoted by the alternation operator is not, as this gives more control to tactic programs. It provides an angelic choice that is implemented through backtracking: on failure, law applications are undone up to the last point where further alternatives are available (as in $t_1 \mid t_2$) and can be explored.

Consider, for example, the following tactic.

```
Tactic promoteVarsExt()  $\hat{=}$   
(law prom-var-state() | law prom-var-state-2()); tactic T()  
end
```

If `law prom-var-state()` succeeds, but `tactic T()` subsequently does not, there is no point in backtracking to apply `law prom-var-state-2()`, and then try `tactic T()` again. Instead, we should cut the search and define the tactic as:

```
!(law prom-var-state() | law prom-var-state-2()); tactic T()
```

Recursive application is supported through the μ_T operator. Sometimes recursive tactics can result in nontermination (**abort**). For practical reasons, our implementation provides bounded forms of recursion (see Section 4.2).

Often, we want to apply tactics to parts of a program; this is supported by structural tactic combinators. Essentially, there is one for each *Circus* program, process, or action constructor. For all the structural combinators, if the application of a tactic to a component program, process, or action fails or aborts, then so does the application of the whole tactic.

4 CRefine Extension

CRefine has been used to automate the application of the *Circus* refinement calculus. However, to reduce time and effort in this application, we developed a new module in CRefine that makes it possible to create tactics and use them in a program development. This extension is describe in the sequel.

4.1 Using Tactics

CRefine's GUI was modified to allow the use of refinement tactics. The changes include the addition of a new item to the menu, *ArcAngelC*, which gives access to a tactic editor, in which users can create, edit and delete tactics. Tactics are written in \LaTeX using specific commands for *ArcAngelC*'s constructs. The

successful compilation results in the addition of the tactic to CRefine. The tactic may then be used as a single transformation rule.

Tactic application can be achieved in the same way as for law application: term selection followed by the tactic selection, possibly with the input of tactic arguments. A successful application results in the update of all frames.

4.2 CRefine's Extended Architecture

The extension to the original architecture of CRefine did not require integration with any further external frameworks. However, new components were added to the tool's architecture. In the presentation layer, we implemented three Java classes and dozens of methods in existing classes to provide the interface of this extension (as presented in the previous section). In the management layer, we did not implement any Java class, but we also extended with 87 new methods that provides support for tactic application following CRefine's initial architecture.

In this extension, we added a new package, *Tactics*, to the data layer. It is responsible for creating and applying refinement tactics and only interacts with the IM (Internal Manager), which is the main class of the management layer. The IM is responsible for controlling the application internally. Hence, the *Tactics* package, through the IM, has access to all other tool packages in order to perform the management of the refinement tactics.

In the *Tactics* package, we added an *ArcAngelC* parser that directly follows the *ArcAngelC* syntax. Furthermore, this package has a sub-package, *Apply*, that provides support for tactics application. In this package each *ArcAngelC* construct was transformed into a Java class that inherits (possibly indirectly) from *TacticComponent*, an abstract class with a single method, *apply*. Most of the classes implement this method following directly the *ArcAngelC* semantics [13]. The only exception is the recursive tactic as we explain in the sequel.

In [13], recursion is defined as the least upper bound of approximation chains (Kleenes theorem). Its direct translation yields a non-terminating calculation because the set of such chains is potentially infinite. In *ArcAngelC*, we use a lazy application of the tactics based on an analysis of the abstract syntax tree, which takes into consideration only those *ArcAngelC* constructs that have influence on the execution paths of a tactic: sequence, alternation, and cut.

For pragmatic reasons, we introduce a tactical for recursive tactics that imposes an upper limit of unfoldings that are performed. The tactical μ_A monitors the number of iterations performed and behaves like **abort** if a certain threshold n is reached. Pragmatically, this supports the implementation of more robust mechanisms for error-catching, as well as the possibility to safely utilise tactics that may fail to terminate. This is especially useful when it is not evident if, and under what conditions, tactics are guaranteed to terminate. By choosing a sufficiently high value for n we obtain a reasonable approximation of the behaviour of the recursive tactic. If we do not want to treat non-termination as an abnormal case, but use it to control the behaviour of other tactics, the alternative tactical μ_F yields failure rather than abortion when the threshold is exceeded.

The user might create his own tactics. For that, he writes the tactic using the specific editor and compiles it. Internally, this compilation involves a lexical and syntactical analysis. This process validates the tactic declaration and their arguments. Currently, predicates, integers, refinement laws, tactic, lists and functions can be given as arguments to tactics. The tactic compilation also checks the existence of refinement laws and tactics used. A successful compilation stores the tactic in the tool's tactic repository. CRefine's current distribution comes with a repository that contains 134 refinement laws (from [13] and [10]).

The tactic application starts with the user selecting a term. The *Tactic* module provides CRefine with a list of tactics that is displayed to the user. We partially implemented a filter of applicable tactics according to the part of the program that is selected. This filter only checks if the first level refinement tactic can be applied to the selected term. When a tactic is chosen (possibly with the input of its arguments, if needed) both the tactic and the term are sent to the IM, which invokes the method `apply` of the tactic giving the selected term as argument. Each implementation of the method `apply` verifies if the selected term fits the structure expected by the tactic. For instance, the Java class that corresponds to the tactic $t_1 \square t_2$, verifies if the selected term is a sequential composition of either *Circus* actions or processes. If this is the case, the tactics t_1 and t_2 are applied to each part of the sequential composition. The result is used by CRefine to update all frames accordingly. If the selected term is not a sequential composition, the tactic application fails.

The implementation of the new module provided CRefine users with refinement tactics, which have been proposed as a means to optimise the application of the *Circus* refinement calculus. Nevertheless, the benefits and scalability of the new module could only be validated after its application to an industrial case study. The next section describes this case study. Its complexity and existent formalisation in *ArcAngelC* [13] made it an excellent candidate with which we were able to validate CRefine's extension providing empirical evidence of its usefulness in practical applications.

5 Case Study

Control systems are often used in safety-critical applications and their verification has been of great interest. In [3], Cavalcanti *et al.* present an approach that aims at proof of correctness of code, as opposed to validation of requirements or designs. They give a semantics to discrete-time Simulink diagrams [1] using *Circus*, and propose a verification technique for parallel Ada implementations.

Control diagrams model systems as directed graphs of blocks interconnected by wires. Simulink [1] is a tool for drawing and analysing such diagrams.

The strategy of this case study is to verify SPARK Ada programs with respect to Simulink diagrams using *Circus*. This verification is based on calculating a *Circus* model for the diagram and calculating a *Circus* model for the SPARK Ada program, and proving that the former is refined by the latter. In the *Circus* model of the diagram, each block is represented by a process, and the diagram by a parallel composition of such processes [13].

Table 1. Case Study

| | NB Phase | BJ Phase | Total |
|----------------------------|----------|----------|-------|
| Tactics of Refinement | 31 | 54 | 85 |
| Tactic Applications | 235 | 203 | 438 |
| Law Applications | 184 | 101 | 285 |
| Proof Obligations | 87 | 50 | 137 |
| Proved Proof Obligations | 81 | 46 | 127 |
| Unproved Proof Obligations | 6 | 4 | 10 |
| ArcAngel Operators | 97 | 78 | 175 |
| Execution Time (seconds) | 7 | 5 | 12 |

The refinement proof is achieved using a refinement strategy that comprises four phases. In [13], we formalise the first two phases: NB and BJ. We split each phase into small steps that correspond to its informal specification presented in [3]. For instance, the phase NB is split into steps NBStep1 to NBStep8. The remaining phases are still in the process of formalization of their tactics.

Using CRefine, we mechanised the first two phases, NB and BJ, and applied it to all blocks of the case study presented. We are currently working on the formalisation (and mechanisation) of the remaining phases.

The NB result was accomplished in 7 seconds in a machine with a Core 2 Duo processor and 4GB RAM. The tactic application generated 87 POs, 81 of which were automatically discharged. The 6 OPs that were not automatically discharged are related to absence of deadlock in *Circus* specifications. The proof of these POs requires the integration with the *Circus* model checker, which is in our Agenda of development. The NB phase includes 31 tactics. The overall application of this phase has 184 refinement law applications. The application of the phase BJ was accomplished in 5 seconds. This phase generated 50 OPs, 46 of which were automatically discharged. The remaining OPs are also related to absence of deadlock in *Circus* specifications. The phase BJ has 54 refinement tactics, 203 refinement tactics applications and 101 refinement laws applications. These results are summarized in Table 1.

This case study can be considered as a good test to check our module for scalability, reliability and runtime. This is due to the fact that it has a large number of refinement law and tactic applications, several ArcAngelC's constructors and the discharge of more than one hundred OPs, whose management are not trivial. In the case study, 64 refinement tactics were successfully defined and used. Together, these tactic exercise the vast majority of ArcAngelC constructs like basic tactics, tacticals, various structural combinators, and program tactics. The remaining constructors were tested individually with unit tests.

One of the contributions in the application of this case study is the automation of 438 refinement tactic applications, all of which were transparent to the user whose only task is to select the overall specification and select the tactics NB and BJ. Furthermore, the refinement laws applications generated 137 OPs; from these, 127 were automatically discharged. As described in [13], despite being

apparently applicable only for a small subset of systems, the tactics developed in our case study is applicable to a very large family of systems developed using Simulink and Ada. This is due to the fact that the tactics presented in [13] and mechanised here cover the vast majority of the block structures found in Simulink Diagrams in practice. Hence, our mechanisation becomes an important contribution to the state-of-the-art in the development of safety-critical applications and their verification (based on [3]).

6 Conclusion

In this paper, we present an extension to CRefine, a tactic module, that allows the definition and use of refinement tactics in a program development as a single transformation rule. We have presented the concepts of the tool and briefly discussed its user interface. CRefine tactics are defined using a refinement-tactic language for *Circus* programs, *ArcAngelC*. These tactics can be used in several developments or even many times within a single development, optimizing the *Circus* refinement calculus. This approach is useful for automating formal development and verification of system models.

The extension of CRefine has been validated using an industrial case study, which consists in the application of a refinement strategy to verify SPARK Ada programs with respect to Simulink diagrams using *Circus*. We have mechanised the first two phase of this strategy, NB and BJ, and applied to all components of the PID controller. This case study involved 21 *ArcAngelC* constructs that include basic tactics, tacticals, various structural combinators, and program tactics. The remaining 18 constructs have been tested with unit tests. The overall application of the case study has 85 refinement tactics, 438 refinement tactics applications, 285 refinement laws applications and 137 POs (127 were automatically discharged). These numbers provide empirical evidence of the applicability (and advantages) of the strategy and the tool support presented here.

Despite being a relatively small example, the case study was proposed by QinetiQ, and its implementation is representative of the architectural pattern used for the development of their safety-critical applications in avionics. Their verification already uses Z and CSP independently to check different aspects of these systems, namely, functionality and scheduling, separately. *Circus* and the refinement strategy that we formalise allows the verification of those aspects as part of a single formal argument. The refinement technique has been developed in conjunction with QinetiQ. With the use of *Circus*, we have managed to enlarge the set of properties and systems that can be checked, without increasing the proof burden, and therefore, the costs. QinetiQ intends to use our strategy (and tools) in the verification of some of their safety-critical systems. Nevertheless, further improvements are in our research agenda as we discuss in the sequel.

The current tactic editor accepts only ASCII characters; users must type the corresponding \LaTeX commands to create and edit their tactics. In a near future, we intend to provide a Unicode editor in which users may use the non-ascii symbols of *ArcAngelC*, like for instance, the structural combinators.

We have only partially implemented the tactics filter used to display available tactics to the developer. In principle, a full implementation of this feature requires an application of all available tactics to the term. However, this approach may become impractical due to the tactics complexity. We will investigate optimised ways to provide this feature without affecting the tool's performance.

The error messages in tactic application need improvement since they do not indicate precisely the source of the errors: when a tactic application fails, a simple error message is given to the user. We intend to improve the messages by creating a log that accurately reports the reasons of the errors.

CRefine intends to be part of a development framework for *Circus* users. That means that using CRefine, users will be able to develop executable code from an abstract *Circus* specification. For that, an integration of all *Circus* tool initiatives is needed. Besides CRefine, the *Circus* model-checker and theorem prover [7], the *Circus* type-checker [16], an improved version of the *Circus* code generator presented in [6], and a *Circus* animator will be part of the *Circus* framework. This integration will provide *Circus* with a complete IDE that will foster the use of *Circus* for software and hardware development.

Acknowledgments. Alessandro Gurgel initially worked on CRefine and helped in the tool's extension. Leo Freitas has provided insights related to the CZT. INES and CNPq partially supports the work of Marcel Oliveira: grants 573964/2008-4, 476836/2009-3, 560014/2010-4. The work of Madiel is supported by CAPES. The COMPASS project (FP7- ICT Call 7 IP) also collaborated with the work presented here.

A Laws of Refinement

Law 1 (prom-var-state). $\text{begin (state } S) \mathbf{L}(x : T) \bullet (\text{var } x : T \bullet \mathbf{MA}) \text{ end}$
 $= \text{begin (state } S \wedge [x : T]) \mathbf{L}(_)\bullet \mathbf{MA} \text{ end}$

Law 2 (prom-var-state-2). $\text{begin } \mathbf{L}(x : T) \bullet (\text{var } x : T \bullet \mathbf{MA}) \text{ end}$
 $= \text{begin (state } [x : T]) \mathbf{L}(_)\bullet \mathbf{MA} \text{ end}$

References

1. Beucher, O.: MATLAB und Simulink (Scientific Computing). Pearson Studium (August 2006)
2. Bouton, T., de Oliveira, D.C.B., Déharbe, D., Fontaine, P.: veriT: An Open, Trustable and Efficient SMT-Solver. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 151–156. Springer, Heidelberg (2009)
3. Cavalcanti, A.L.C., Clayton, P., O'Halloran, C.: From Control Law Diagrams to Ada via *Circus*. Formal Aspects of Computing (2011) (online first)
4. Cavalcanti, A.L.C., Sampaio, A.C.A., Woodcock, J.C.P.: A Refinement Strategy for *Circus*. Formal Aspects of Computing 15(2–3), 146–181 (2003)
5. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)

6. Freitas, A., Cavalcanti, A.L.C.: Automatic Translation from Circus to Java. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 115–130. Springer, Heidelberg (2006)
7. Freitas, L., Cavalcanti, A.L.C., Woodcock, J.C.P.: Taking Our Own Medicine: Applying the Refinement Calculus to State-Rich Refinement Model Checking. In: Liu, Z., Kleinberg, R.D. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 697–716. Springer, Heidelberg (2006)
8. Gurgel, A.C., de Medeiros Jr., V.G., Oliveira, M.V.M., Déharbe, D.B.P.: Integrating SMT-Solvers in Z and B Tools. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 412–413. Springer, Heidelberg (2010)
9. Martin, A.P., Gardiner, P.H.B., Woodcock, J.C.P.: A Tactical Calculus. *Formal Aspects of Computing* 8(4), 479–489 (1996)
10. Oliveira, M.V.M.: Formal Derivation of State-Rich Reactive Programs using Circus. PhD thesis, Department of Computer Science, University of York (2006)
11. Oliveira, M.V.M., Cavalcanti, A.L.C., Woodcock, J.C.P.: ArcAngel: a Tactic Language for Refinement. *Formal Aspects of Computing* 15(1), 28–47 (2003)
12. Oliveira, M.V.M., Gurgel, A.C., de Castro, C.G.: CRefine: Support for the Circus Refinement Calculus. In: Cerone, A., Gruner, S. (eds.) 6th IEEE International Conferences on Software Engineering and Formal Methods, pp. 281–290. IEEE Computer Society Press (2008)
13. Oliveira, M.V.M., Zeyda, F., Cavalcanti, A.L.C.: A Tactic Language for Refinement of State-rich Concurrent Specifications. *Science of Computer Programming* 76(9), 792–833 (2011)
14. Roscoe, A.W.: *The Theory and Practice of Concurrency*. Prentice-Hall. Series in Computer Science. Prentice-Hall (1998)
15. Woodcock, J.C.P., Davies, J.: *Using Z—Specification, Refinement, and Proof*. Prentice-Hall (1996)
16. Xavier, M.A., Cavalcanti, A.L.C., Sampaio, A.C.A.: Type Checking Circus Specifications. In: Moreira, A.M., Ribeiro, L. (eds.) SBMF 2006: Brazilian Symposium on Formal Methods, pp. 105–120 (2006)

JSXM: A Tool for Automated Test Generation

Dimitris Dranidis¹, Konstantinos Bratanis², and Florentin Ipate³

¹ The University of Sheffield International Faculty, CITY College,
Computer Science Department,
3 Leontos Sofou, 54626, Thessaloniki, Greece
dranidis@city.academic.gr

² South-East European Research Centre (SEERC),
International Faculty, The University of Sheffield,
24 Proxenou Koromila, 54622, Thessaloniki, Greece
kobratanis@seerc.org

³ University of Pitesti,
Department of Computer Science and Mathematics,
Str. Targu din Vale 1, 110040 Pitesti, Romania
florentin.ipate@ifsoft.ro

Abstract. The Stream X-machine (SXM) is an intuitive and powerful modelling formalism that extends finite state machines with a memory (data) structure and function-labelled transitions. One of the main strengths of the SXM is its associated testing strategy: this *guarantees* that, under well defined conditions, *all* functional inconsistencies between the system under test and the model are revealed. Unfortunately, despite the evident strength of SXM based testing, no tool that convincingly implements this strategy exists. This paper presents such a tool, called JSXM. The JSXM tool supports the animation of SXM models for the purpose of model validation, the automatic generation of abstract test cases from SXM specifications and the transformation of abstract test cases into concrete test cases in the implementation language of the system under test. A special characteristic of the modelling language and of the tool is that it supports the specifications of flat SXM models as well as the integration of interacting SXM models.

Keywords: model-based testing, automated test generation, functional conformance, incremental testing, stream x-machines, implementation, tool.

1 Introduction

Performing systematic software testing manually is a time consuming and costly process, because it requires the manual definition, maintenance and execution of appropriate test cases. The manual definition of test cases can be increasingly complex, often resulting in test cases that may not provide full coverage of the implementation. Also, manual software testing is prone to human errors, which may be introduced either in the selection or execution of test cases. Consequently,

there is a clear need for automating software testing. The automation increases the reliability of the software testing process, because there is minimum human involvement in the generation and the execution of test cases.

One approach to the automation of software testing is model-based testing (MBT). MBT enables the automatic generation of test cases using models that specify the expected behaviour of the implementation under test (IUT). If MBT relies on a formalism for behavioural modelling and test case generation, various properties can be guaranteed for the IUT, such as functional conformance. One of the popular approaches to MBT is based on state-based descriptions, which are typically used to model the control flow of a system.

Stream X-machine (SXM) [9] is a state-based formalism capable of modelling both the data and the control of a system. SXMs are special instances of the X-machines introduced by Eilenberg [8]. SXMs extend finite state machines by incorporating memory and processing functions instead of simple labels. The powerful modelling capabilities of SXMs have been used in a number of research projects, such as the EURACE, SUMO and Epitheleome Project¹, for the simulation of cellular and social systems. Furthermore, SXMs have the significant advantage of offering a testing method [12,9] that, under certain design-for-test conditions, ensures the conformance of an IUT to its specification.

Although there have been several improvements to the SXM testing method with the aim to relax the design-for-test conditions [11,13], there exists no tool that demonstrates the practical benefits of the method. The small number of existing tools [16,17,15] are concerned with the modelling and the animation of SXM models, but not with the use of SXMs for automated software testing.

In this paper, we present JSXM, a new (and, to the best of our knowledge, the only existing) suite of tools that supports automated model-based test generation using SXMs. JSXM supports animation of SXM models, model-based test generation and test transformation. The test generation is based on the SXM testing method and, given an SXM model, it generates a set of test cases in XML format, which are independent of the programming language of the implementation. Test transformation is used for transforming the general test cases to concrete test cases in the underlying technology of the implementation. Currently, a JUnit transformer is available¹, which generates JUnit test cases.

The rest of the paper is organised as follows. Section 2 presents the SXM formalism and Section 3 the SXM testing method and the improvements of the method to relax the design-for-test conditions. Section 4 presents JSXM and describes how the SXM testing method has been implemented. Section 5 discusses the evaluation of JSXM in terms of effectiveness of the generated test cases, efficiency of the test generation process and the validation of JSXM in existing applications. Section 6 outlines the existing related tools for SXMs. Finally, Section 7 concludes the paper and presents some future directions.

¹ <http://www.flame.ac.uk>

2 Stream X-Machines

A Stream X-machine [9] is a computational state-based model capable of modelling both the data and the control of a system. SXMs extend finite state machines with two important additions: (a) the machine has some internal storage, called *memory*; and (b) the transition labels are not simple symbols, but *processing functions* that represent basic operations that the machine can perform. A processing function processes inputs from an input stream and produces outputs on an output stream while it may also change the value of the memory.

Definition 1. An SXM is a tuple $Z = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ where:

- Σ is a finite set of input symbols and Γ is a finite set of output symbols;
- Q is a finite set of states;
- M is a (possibly) infinite set called memory;
- Φ is a finite set of partial functions ϕ (called processing functions) that map memory-input pairs to output-memory pairs, $\phi : M \times \Sigma \rightarrow \Gamma \times M$;
- F is the next-state partial function, $F : Q \times \Phi \rightarrow Q$;
- $q_0 \in Q$ and $m_0 \in M$ are the initial state and initial memory respectively.

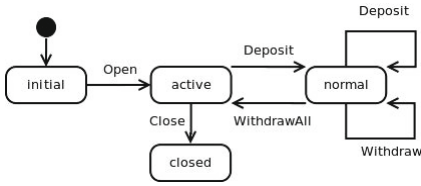
An SXM is usually graphically represented by the state diagram of its associated finite state automaton: $A_Z = (\Phi, Q, F, q_0)$.

Example 1. Fig. 1 illustrates the associated automaton of an SXM model of a system that allows performing basic operations on a bank account. For simplicity, let us assume that the system’s interface comprises four operations: *open()*, *deposit(a)*, *withdraw(a)*, and *close()*, where a is a positive number. These comprise the input alphabet Σ of the SXM. The memory of the system consists of a single number, the available balance b . Initially an account is inactive (state: *initial*) and needs to be opened (state: *active*) with its balance set to zero before any transaction can be performed. Depositing an amount results in increasing the balance (state: *normal*), while the withdrawal of an amount can only take place if the amount does not exceed the balance. An account can be closed only if its balance is zero and once closed (state: *closed*) it cannot be re-activated. The above requirements are specified in the state diagram and in the definition of its processing functions (shown on the right of the diagram in Fig. 1).

The next state function F can be extended to the function $F^* : Q \times \Phi^* \rightarrow Q$ receiving sequences of processing functions (paths in the associated automaton). The language accepted by the associated automaton of Z from state q is defined as $L_{A_Z}(q) = \{p \in \Phi^* \mid (q, p) \in \text{dom } F^*\}$ and the language accepted by the automaton is notated as $L_{A_Z} = L_{A_Z}(q_0)$.

An SXM is *deterministic* if for every state, memory, input combination there is at most one possible transition, i.e. for every $\phi_1, \phi_2 \in \Phi$ such that $(q, \phi_1) \in \text{dom } F$ and $(q, \phi_2) \in \text{dom } F$ for some $q \in Q$ either $\phi_1 = \phi_2$ or $\text{dom } \phi_1 \cap \text{dom } \phi_2 = \emptyset$. In this paper we only consider deterministic SXMs. It can be checked that the

² Processing function names are capitalized to easily distinguish from input symbols.



$$\begin{aligned}
 Open(b, open()) &= (openOut, 0) \\
 Deposit(b, deposit(a)) &= (depositOut, b + a) \\
 Withdraw(b, withdraw(a)) &= (withdrawOut, b - a) \text{ if } a < b \\
 WithdrawAll(b, withdraw(a)) &= (withdrawOut, 0) \text{ if } a = b \\
 Close(b, close()) &= (closeOut, b)
 \end{aligned}$$

Fig. 1. A state transition diagram for an SXM modelling a bank account and its processing functions (e.g., *Close*), which are triggered by input symbols (e.g., *close()*)

account SXM is deterministic, since *Withdraw* and *WithdrawAll*, which both accept the input *withdraw(a)* at state *normal*, have disjoint domains.

If $p \in \Phi^*$ is a sequence of processing functions then the function $\|p\| : M \times \Sigma^* \rightarrow \Gamma^* \times M$ associates a memory value and a stream of input symbols with the corresponding stream of output and final memory produced by following p .

An SXM Z is *completely-defined* if every sequence of inputs $s \in \Sigma^*$ is processed by at least one sequence of functions $p \in L_{AZ}$ accepted by the associated automaton. Any SXM that is not completely defined can be transformed into one that is completely defined by adding at each state and for each not accepted input σ a self-transition labelled with the processing function σ_{error} . To make the account SXM completely defined we need to add the processing functions: $Open_{error}, Deposit_{error}, Withdraw_{error}, Close_{error}$, and use these as labels for loop transitions at the states that the corresponding inputs are refused. For simplicity these transitions are not shown in the diagram.

3 The SXM Testing Method

SXMs have the significant advantage of offering a testing method that under certain design-for-test conditions ensures the conformance of an IUT to a specification. The goal of the testing method is to devise a finite test set $X \subseteq \Sigma^*$ that produces identical results when applied to the specification and the IUT *only if* they both compute identical functions. The main assumption that needs to be made for the IUT is that it consists of correct elementary components, i.e. the processing functions are correctly implemented. Furthermore, it is estimated that the number of states in the IUT is $n' \geq n$, where n is the number of states of the specification. Let $k = n' - n$.

There have been several improvements to the method with the aim to relax the design-for-test conditions. The main variants are briefly presented next.

3.1 Test Generation for Input-Complete Specifications

The original SXM testing method [12,9] relies on the following design-for-test conditions:

- **Output-distinguishability:** Processing functions should be distinguishable by their different outputs on some memory-input pair, i.e. for every

$\phi_1, \phi_2 \in \Phi$, $m \in M$ and $\sigma \in \Sigma$ such that $(m, \sigma) \in \text{dom } \phi_1$ and $(m, \sigma) \in \text{dom } \phi_2$, if $\phi_1(m, \sigma) = (\gamma, m_1)$ and $\phi_2(m, \sigma) = (\gamma, m_2)$ then $\phi_1 = \phi_2$.

- **Controllability:** Processing functions have to be input-complete w.r.t. memory, i.e. for each $m \in M$ there is $\sigma \in \Sigma$ such that $(m, \sigma) \in \text{dom } \phi$. This implies that for all memory values there should exist an input value that triggers the execution of the processing function.

The test generation is a two stage process: (1) the W method [2] is applied on the associated automaton A_Z to produce a set $Y \subseteq \Phi^*$ of sequences of processing functions, which are then (2) translated into sequences of inputs for Z .

Y is obtained by constructing a state cover S and a characterization set W of A_Z . $S \subseteq \Phi^*$ contains sequences to reach all states of A_Z , while $W \subseteq \Phi^*$ contains sequences to distinguish between any two distinct states of A_Z . Each sequence $y \in Y$ consists of three sub-sequences, i.e., $y = stw$, where $s \in S$ drives the automaton to a specific state, $t \in \Phi^*$ attempts to exercise transition-paths up to length of $k+1$ and w distinguishes the resulting state from any other state. Thus $Y = S\Phi[k+1]W = S(\bigcup_{0 \leq i \leq k+1} \Phi^i)W$. Note that Y may include legal as well as non-legal sequences of processing functions (positive and negative testing).

Definition 2. A test function $t : \Phi^* \rightarrow \Sigma^*$ is a function that converts a sequence of processing functions to a sequence of inputs and is defined as:

$$\begin{aligned} - t(\epsilon) &= \epsilon \\ - t(p\phi) &= \begin{cases} t(p) & \text{if } p \notin L_{A_Z} \\ t(p)\sigma & \text{if } p \in L_{A_Z} \end{cases} \end{aligned}$$

where $p \in \Phi^*$ is a sequence of processing functions, $\phi \in \Phi$, and $\sigma \in \Sigma$ is a suitable input such that $(m, \sigma) \in \text{dom } \phi$ and $\|p\|(m_0, t(p)) = (g, m)$, i.e. m is the attained memory after executing the path p .

Thus, for a path that belongs to the language of the associated automaton, a sequence of inputs of the same length is produced. For a path that does not belong to the language of the associated automaton, the generated input sequence is one input longer than the longest accepted prefix of the sequence. The extra input attempts to exercise the first non-existing transition. Such an input is guaranteed to exist for every processing function if the machine is controllable.

The final test suite for checking functional equivalence is:

$$X = t(Y) = t(S\Phi[k+1]W)$$

The controllability condition is rather strict and in most practical situations specifications are not naturally input-complete. The account SXM is not controllable since *Withdraw* and *WithdrawAll* are only defined for $m > 0$ and $m > 1$ respectively. If a specification is not input-complete then the specification as well as the IUT needs to be augmented with extra inputs. However, the introduction of the extra inputs might lead to several problems, since after successful testing, the extra inputs of the IUT need to be removed or hidden, with the potential danger of introducing faults to the IUT. In order to alleviate this problem, the testing method has been generalized in [11] to non-controllable specifications, as presented in the following section.

3.2 Test Generation for Input-Uniform Specifications

In [11] the notion of realizable sequences is introduced. There may exist sequences of functions that are accepted by the automaton but they cannot be driven by any input sequence. These sequences are called non-realizable.

Definition 3. A sequence $p \in \Phi^*$ is called realizable in q and m if $p \in L_{AZ}(q)$ and $\exists s \in \Sigma^*$ such that $(m, s) \in \text{dom } \|p\|$. The set of realizable sequences of Z in q and m is notated as $LR_Z(q, m)$. Let LR_Z be defined as $LR_Z(q_0, m_0)$.

A state is r -reachable if it can be reached by a realizable sequence $p \in LR_Z$.

Definition 4. A set $S_r \subseteq LR_Z$ is called a r -state cover of Z if for every r -reachable state q of Z there exists a unique $p \in S_r$ that reaches the state q .

The set of memory values that can be attained at a state q is notated as $MAtt(q)$ and it consists of all memory values that are the result of realizable sequences that end at state q , i.e. $MAtt(q) = \{m \in M \mid \exists p \in LR_Z \text{ and } \exists s \in \Sigma^*, \|p\|(m_0, s) = (q, m)\}$.

Example 2. In the account SXM, all sequences of processing functions that are accepted by the associated automaton are also realizable. $S_r = \{\epsilon, \text{Open}, \text{Open Deposit}, \text{Open Close}\}$. At states *initial*, *active*, and *closed* the only attainable memory is $m = 0$; at state *normal*, $m > 0$.

Definition 5. A set $Y \subseteq \Phi^*$ r -distinguishes between q_1 and q_2 if for every $m_1 \in MAtt(q_1)$ and every $m_2 \in MAtt(q_2)$ there exist sequences in Y that are realizable either in m_1 and q_1 or in m_2 and q_2 but not in both of them, i.e. $LR_Z(q_1, m_1) \cap Y \neq LR_Z(q_2, m_2) \cap Y$.

The proposed testing method has replaced the controllability condition with a less strict design-for-test condition that requires the specification to be *input-uniform*. When a specification is input-uniform, the inputs that will drive a sequence of functions can be selected one at a time. If a sequence ϕ_1, \dots, ϕ_n is realizable and the specification is input-uniform and $\sigma_1, \dots, \sigma_{n-1}$ is any input sequence that drives the sequence $\phi_1, \dots, \phi_{n-1}$ then there always exist an input σ_n that will drive the next function ϕ_n . This implies that the specific choices of the previous inputs do not influence the choice of the inputs that follow.

Since the model is not required to be controllable the test generation method cannot rely on the W method. It is based on a state counting approach involving the construction of the cross-product machine of the specification and the IUT.

Example 3. The account SXM is not input-uniform as it can be illustrated with the following example: Although the sequence *Open Deposit Withdraw* is realizable, for instance with the sequence: *open() deposit(2) withdraw(1)*, a different choice of inputs for *Open Deposit* as *open() deposit(1)* does not allow to find any input for *Withdraw*, since the balance is 1 and withdrawing the least amount of 1 is not possible (the guard condition for *Withdraw* is $a < b$). The *withdraw(1)* input will fire the *WithdrawAll* function instead.

3.3 Test Generation for Generic Specifications

An even more general approach was proposed in [13], in which the only required design-for-test condition is output-distinguishability. The test generation method is also based on state counting using the cross-product automaton of the specification and the implementation.

On the other hand, the condition required for distinguishing states has been strengthened: two states will have to be *separable*, i.e. distinguished by two realizable sequences with *overlapping* domains.

Definition 6. A pair of states (q_1, q_2) is separable if there exists a finite set of sequences Y such that $\forall m_1 \in MAtt(q_1), m_2 \in MAtt(q_2)$, there exists $p_1 \in LR(q_1, m_1) \cap Y$ and $p_2 \in LR(q_2, m_2) \cap Y$ such that $dom p_1 \cap dom p_2 \neq \emptyset$.

Definition 7. A set $W_s \subseteq \Phi^*$ is called a separating set of Z if it separates (distinguishes) between every pair of separable states of Z .

Example 4. The state *initial* in the account SXM is separable from all other states since the sequences *Open* and *Open_{error}* have overlapping domains; at any m they accept the sequence *open()*. A separating set that separates all state pairs for the account SXM is: $W_s = \{Open, Open_{error}, Close, Close_{error}, WithdrawAll, Withdraw_{error}\}$.

An important theoretical result presented in [13] involves the case in which S_r reaches all states of Z and W_s separates all pairs of states in Z . In that special case the testing method reduces to a variant of the W -method:

$$Y = UW_s = ((S_r\Phi[k+1]) \cap L_{A_Z})W_s$$

Furthermore, the testing method requires that all sequences of $U = (S_r\Phi[k+1]) \cap L_{A_Z}$ are realizable, i.e. it is required that $U \subseteq LR_Z$. Note that the sequences of processing functions of maximum length $k+1$ that follow the r -state cover are limited to those that are accepted by the associated automaton.

In the SXM account example the set S_r covers all states, the set W_s distinguishes between all pairs of states and U is realizable, therefore the testing method can be applied for functional equivalence.

4 The JSXM Tool

JSXM [3] is a tool, developed in Java, that allows the specification of SXM models, their animation and most importantly the automated test generation.

Animation of the model means the execution of the model by providing an input stream and observing the resulting output stream. The interactive or batch animation, which are supported by the tool, allow the model designer to validate the specification, i.e. ensure that the correct functionality is modelled.

Once a model is validated, it can be used for the automated generation of test cases. The test cases that are generated by JSXM are in XML format and they

are independent of the technology or programming language of the implementation. These general test cases can then be transformed by the JSXM tool to test cases in the programming languages of the IUT.

In the following sections we briefly describe the JSXM modelling language and the associated tool suite.³

4.1 The JSXM Modelling Language

The JSXM modelling language is an XML-based language with Java in-line code. This allows software engineers who are familiar with these widespread technologies to model systems in a formalism that allows the automated generation of test cases and removes the barrier of having to learn a new notation.

The states and the transitions are described in XML. An extract of the JSXM code⁴ for representing the state transition diagram of Fig. 11 is provided below:

```
<states>
  <state name="initial" /><state name="active" />
  <state name="closed" /> <state name="normal" />
</states>
<initialState state="initial" />
<transitions>
<transition from="initial" function="Open" to="active" />
  <transition from="active" function="Close" to="closed" />
  <transition from="active" function="Deposit" to="normal" />
  ...
</transitions>
```

The input and the output symbols are also described in XML code. Input and output symbols can be basic symbols (such as *openRequest*) or compound symbols carrying arguments, as for example the *deposit(amount)* input that carries an integer argument. The types of the arguments are specified as XSD types. The modeller can extend the types with any user-defined complex XSD type. The outputs are structured in a similar way to inputs:

```
<inputs>
  <input name="open" />
  <input name="close" />
  <input name="deposit">
    <arg name="amount" type="xs:int" />
  </input>
  <input name="withdraw">
    <arg name="amount" type="xs:int" />
  </input>
</inputs>
<outputs>
  <output name="openOut" />
  <output name="closeOut" />
  <output name="depositOut">
    <result name="amount" type="xs:int" />
  </output>
  <output name="withdrawOut">
    <result name="amount" type="xs:int" />
  </output>
</outputs>
```

The memory and the body of the processing functions are written in in-line Java code. This allows the definition of any complex Java data structure as the memory of the system.

```
<memory>
  <declaration> int balance </declaration> <initial> balance = 0 </initial>
</memory>
```

³ The tool can be downloaded from <http://www.jsxm.org>

⁴ The complete JSXM specification for the SXM account can be found at <http://www.jsxm.org/SEFM2012>

Processing functions are specified by defining their inputs, outputs, preconditions (specifying the domain of the function) and effects on the memory. For brevity only one processing function is shown. Note the *object.get_par()* approach for retrieving the parameter *par* of compound inputs and outputs.

```
<function name="Withdraw" input="withdraw" output="withdrawOut">
  <precondition> balance > withdraw.get_amount() </precondition>
  <effect>
    balance = balance - withdraw.get_amount();
    withdrawOut.set_amount(withdraw.get_amount());
  </effect>
</function>
```

4.2 Test Generation Process

The JSXM tool implements the extended *W*-method for generic specifications (Section 3.3) for the generation of the test set. For the test generation process the modeller needs to provide:

- a JSXM specification of the SXM model Z .
- an r -state cover S_r and a separating set W_s
- a set of input generators (explained in detail in Section 4.3).
- the estimated difference k of states between the IUT and the specification.

The test generation process consists of the following steps:

1. It is being checked that all states of Z are r -reachable by S_r .
2. The set $S_r\Phi[k+1]$ is generated.
3. The generated set is restricted to these sequences that are accepted by the associated automaton: $U = (S_r\Phi[k+1]) \cap L_{A_Z}$
4. It is being checked that all generated sequences are realizable, i.e. $U \subseteq LR_Z$.
5. The sequences of the characterization set W are attached to the generated sequences yielding: $UW_s = ((S_r\Phi[k+1]) \cap L_{A_Z})W_s$
6. The resulting set of sequences is optimized: any sequence that is a proper prefix of another sequence in the set is removed from the set, since any faulty behaviour produced by this proper prefix will be identified during the animation of the prefix as part of the longer sequence.
7. All sequences of processing functions are transformed to sequences of inputs generated by the input generators. During this step it is expected that some of the sequences will not be realizable (at their W_s postfix) as formally described in the definition of the test function.
8. Finally, all the input sequences are fed to the animator that acts as an oracle and provides the expected output sequences.

As described in Section 3.3, if W_s separates all pairs of states and all sequences in step 4 are realizable, then the resulting suite is sufficient to guarantee the functional conformance of the implementation to the specification.

4.3 Input Generators

The test function transforms sequences of processing functions to sequences of inputs. The modeller has to define for each processing function an input generator function.

An input generator function is a (partial) function $in_\phi : M \rightarrow \Sigma$ for $\phi \in \Phi$. The input generator guarantees to find an input, if the memory is in the domain of the input generator. Otherwise a suitable input does not exist to make the function fire. We could define the following input generators for the account:

$$\begin{aligned} in_{Deposit}(balance) &= 100 \\ in_{Withdraw}(balance) &= 1 \text{ if } balance > 1 \\ in_{WithdrawAll}(balance) &= balance \text{ if } balance > 0 \end{aligned}$$

The transformation of a sequence $p \in \Phi^*$ of processing functions to a sequence of inputs is recursively defined as:

$$\begin{aligned} - t(\epsilon) &= \epsilon \\ - t(p\phi) &= \begin{cases} t(p)in_\phi(m) & \text{if } p \in LR_Z \text{ and } m \in \text{dom } in_\phi \\ t(p) & \text{otherwise} \end{cases} \end{aligned}$$

where m is the memory attained after the sequence $t(p)$, i.e. $\|p\|(m_0, t(p)) = (g, m)$. From the definition it is clear that it is being assumed that the specification is input-uniform. In what follows we discuss what are the practical limitations of this assumption and how they may be overcome.

If the specification is controllable then the processing functions are input-complete w.r.t. memory, which implies that for every possible memory value there exists an input that triggers the function and therefore in_ϕ can also be totally defined. If the specification is input-uniform and the input generator is defined for all attainable memory values, then the algorithm is able to find suitable input sequences for all realizable sequences of processing functions.

If, however, the specification is not input-uniform, the generation of inputs for realizable sequences is not generally guaranteed as it is illustrated in the following example. The input generators defined above for the account SXM would fail to generate appropriate inputs for the following long but still realizable sequence: *Open Deposit Withdraw*¹⁰⁰, in which 100 *Withdraws* follow a *Deposit*. The sequence could still be made realizable by increasing the amount parameter of the *deposit*(100) input that was generated for *Deposit*. So the input generators successfully generate input values up to a certain length of the function sequence.

The length of the function sequences depends on k . Even in the cases of non input-uniform specifications, the modeller can carefully design the input generators in such a way so that the specification “behaves” uniformly for all bounded-sequences that will be generated for the selected k . In most practical situations k takes small values (usually less than 5). Larger k values are not expected to reveal new errors (unless the implementation is grossly erroneous) and are computationally expensive for producing the test sets, since the number of test cases and the total length of the test set depend exponentially on k .

Nevertheless, we are investigating solutions that will generalise the input generators so that all realizable sequences can be generated. We are currently examining the possibility of applying constraint satisfaction techniques to solve this problem. If we need to generate an input sequence for: *Open Deposit Withdraw Withdraw Withdraw*, the following symbolic input sequence could be generated: $deposit(x) withdraw(w_1) withdraw(w_2) withdraw(w_3)$, with the related arithmetic constraints that need to be solved: $x > 0, w_1 < x, w_1 + w_2 < x, w_1 + w_2 + w_3 < x$. A constraint solver could return the following values: $x = 4, w_1 = w_2 = w_3 = 1$, which would make the sequence realizable.

4.4 Generated Test Cases and Transformation

For the input-output test cases to be produced, all the input sequences are fed to the JSXM animator, which acts as an oracle, and the resulting output sequences are recorded. The resulting test cases (pairs of input and output sequences) are stored in a XML file⁵ in a programming language independent format.

In order to execute the tests on the IUT, the language independent generated test cases need to be transformed to the programming language of the IUT. This is the task of a test transformer. The JSXM tool allows modellers to extend the tool by defining their own test transformer for any programming language. Currently the JSXM test suite offers a JUnit test transformer, which transforms the abstract test cases into JUnit test cases.⁶

4.5 IUT Wrappers

Since the SXM model and the IUT are at different levels of abstraction, a wrapper (or adapter) is needed that will translate IUT values back to model values. In the case of Java testing, a wrapper class consists of wrapper methods for each method of the class under test. All control flow paths within a method should produce some observable result: either a returned value or an exception. These results are then translated by the corresponding wrapper method to values that will be compared by the test engine with the model outputs.

For example, a Java method *public void open()* may perform some processing (change the internal state) but it is not expected to return any result. For the purpose of testing, however, the implementation needs to satisfy the observability design-for-test condition. A wrapper wraps the method call and, depending on the outcome, returns the expected output that will be matched with the output of the model (e.g., *openOut*). Similarly, if the *open* method throws an exception (because it could not be executed) the exception has to be caught by the wrapper and a model output should be returned (e.g., *openError*).

4.6 Interacting JSXM Models and Incremental Testing

By using interacting SXM models one can perform automated incremental testing of classes that are based on other classes.

⁵ The generated XML file can be found at <http://www.jsxm.org/SEFM2012>

⁶ The generated JUnit test cases can be found at <http://www.jsxm.org/SEFM2012>

This is achieved via a special type of parameters in SXM specifications, which is the *sxm* type. Values of this type are *instances* of SXM models. The interaction of SXM instances resembles the object-oriented method invocation: an SXM, within the effect of a processing function, may send a “message”, in the form of an input symbol, to another SXM, which produces an output, which is returned to the sender SXM.

In the following example we present the memory, a processing function and its corresponding input from a borrower SXM. The borrower may borrow a book if a book is available. A separate book SXM model exists, which specifies the behaviour of a book (borrowing, returning, checking availability).

```
<memory>
  <declaration> BookSXM book; </declaration><initial> book = null; </initial>
</memory>
<inputs><input name="borrowBook" ><arg name="book" type="sxm" /></input></inputs>
<functions>
  <function name="BorrowBook" input="borrowBook" output="borrowBookOut" >
    <precondition> ((BookSXM) (borrowBook.get_book())).isAvailable().result;
    </precondition>
    <effect>
      book = (BookSXM) borrowBook.get_book();
      book.setBorrowed();
    </effect>
  </function>
</functions>
```

Through the *BorrowBook* processing function, the borrower SXM instance sends the input *isAvailable()* to a book SXM instance. The book SXM instance is being animated with this input and an output is produced, which is returned to the borrower SXM instance.

Test sets are generated separately for each SXM and its corresponding class and programs can be incrementally tested. This incremental approach removes the need to create a separate flat model which specifies the whole system.

5 Evaluation

5.1 Validation of the JSXM Tool in Existing Applications

The JSXM tool has been used and practically validated in several application scenarios described briefly below. All the applications are from the area of service-oriented computing and Web services. The XML-based specifications of JSXM and the capability to extend the input types by user-defined XSD types facilitate easier integration with Web technologies and related XML-based Web service standards.

In [5] SXMs are utilised for modelling the behavioural specification of Web services and the SXM testing method is used for generating test cases, which verify that the service implementations conform to their specification. This approach is further elaborated in [18] where JSXM is used as the main tool in a framework for the validation and verification of Web services.

The animation capability of the JSXM tool (execution of SXM models) enables the run-time verification approach that is presented in [6]. The animator serves as an oracle that provides the expected outputs for the purpose of run-time monitoring of Web services. JSXM provides an API that allows calling the animator’s methods programmatically. This API is utilised in the implementation of the run-time verification architecture described in [1].

In [7] a novel approach is proposed for just-in-time testing of conversational Web services. The aim of just-in-time testing is to detect potential problems (functional inconsistencies) in service implementations and to pro-actively trigger adaptations of the service-based application. The JSXM and its test generation capability is utilised in order to automatically generate test cases on the fly without any human intervention and to test the service before its invocation.

It is important to notice that in all these applications the SXM testing method, and its realization through the JSXM tool, managed to successfully identify errors in the implementations, even with small values of $k \leq 2$. Furthermore, although the SXM testing method is based on the assumption that the basic processing functions are correctly implemented in the IUT, the generated test set managed to reveal not only control flow errors but also errors in the implementation of the functions. This implies that practically SXM testing is also able to reveal functional mismatches in the processing functions, although this capability is not theoretically guaranteed. This capability is demonstrated in the next section on the example of the account SXM.

5.2 Effectiveness and Efficiency of the Test Generation

To demonstrate the effectiveness of the generated test set we have performed mutation testing on the Java class Account, which is the implementation of the account SXM. Mutation testing was performed with Jumble [14], which is a class level mutation testing tool that works in conjunction with JUnit. The test set generated from JSXM for $k = 0$ managed to kill all 19 mutants⁷. The mutants changed both control logic of the program (negated conditionals, boundary values in conditions) as well as assignments within program paths (arithmetic operators, assigned values).

Concerning the efficiency of the test generation process, Table 1 shows the number of test cases generated and the time required for the test generation for different values of k . The test generation was performed on a Quad Core i7, 2.5 Ghz. The absolute times in seconds are provided only as an indication.

Table 1. Number of generated test cases and generation time for different values of k

| k | 0 | 1 | 2 | 3 |
|------------------------|------|------|------|-------|
| number of test cases | 143 | 841 | 5293 | 35151 |
| time to generate (sec) | 0.06 | 0.26 | 1.67 | 46.06 |

⁷ A description of the mutants can be found at <http://www.jsxm.org/SEFM2012>

6 Related Tools

Currently, a small number of tools is available for SXMs. Most tools are concerned with the modelling and the animation of SXM models.

The Flexible Large-scale Agent Modelling Environment (FLAME) [10] is a framework that uses SXMs for modelling agent-based systems and for generating simulations in C. The FLAME framework provides an XML-based specification language, referred as XMML, for creating specifications that serve as models of the behaviour of agents. In-line C code is used for defining the behaviour of the processing functions, similarly as JSXM uses in-line Java code. FLAME includes Xparser, which is a parser for parsing XMML, in order to automatically generate simulation programs in C that can run models efficiently on HPCs [16].

X-system [15] is a tool that facilitates the modelling and the animation of SXMs. X-system has been implemented in Prolog and uses the X-Machine Definition Language (XMDL) for writing the specifications. XMDL was later extended by XMDL-O [4], which supported an object-based notation. Both languages are supported by the X-system, which is used mainly for animation of models.

Ma et al. [17] describe a tool for SXM models and the automatic generation of tests. It is not evident if this tool is capable of generating concrete test cases (sequences of inputs), since the authors demonstrate only the generation of processing function sequences (first step of the test generation process).

To the best of our knowledge, JSXM is the only tool available for automatically generating *concrete* (executable) test cases based on the SXM testing method. Neither FLAME nor X-system are known to be able to generate concrete test cases.

7 Conclusions

In this paper we have presented the JSXM tool for automated testing. The tool builds on the SXM formalism and implements an extension of the W method for generating test sets that are able to guarantee the functional conformance of an IUT to its specification. Currently the JSXM test generation is guaranteed to work in the case of input-uniform specifications. In Section 4.3, however, we have discussed and demonstrated through an example, that the test generation can practically work even in the case of some non input-uniform specifications.

Additionally, the JSXM language allows the definition of interacting SXM model instances. This feature enables the incremental testing of object oriented programs, where each class is separately modelled as an SXM.

The tool's applicability has been demonstrated in several application scenarios from the area of Web service testing, monitoring and run-time verification.

In the future we intend to change the input generators so that the test generation works for generic specifications. We also intend to implement the test generation based on state-counting, so that the tool covers also cases of specifications in which not all states are separable.

Acknowledgement. The work of F. Ipate was supported by a grant from the Romanian National Authority for Scientific Research, CNCS-UEFISCDI, project number PN-II-ID-PCE-2011-3-0688.

References

1. Bratanis, K., Dranidis, D., Simons, A.J.H.: An extensible architecture for run-time monitoring of conversational web services. In: 3rd International Workshop on Monitoring, Adaptation and Beyond/ECOWS (2010)
2. Chow, T.S.: Testing software design modelled by finite state machines. *IEEE Transactions on Software Engineering* 4, 178–187 (1978)
3. Dranidis, D.: JSXM: A suite of tools for model-based automated test generation: User manual. Tech. rep., Technical Report WP-CS01-09. CITY College (2009)
4. Dranidis, D., Eleftherakis, G., Kefalas, P.: Object-based language for generalized state machines. *Annals of Mathematics, Computing and Teleinformatics (AMCT)* 1(3), 8–17 (2005)
5. Dranidis, D., Kourtesis, D., Ramollari, E.: Formal verification of web service behavioural conformance through testing. *Annals of Mathematics, Computing & Teleinformatics* 1(5), 36–43 (2007)
6. Dranidis, D., Ramollari, E., Kourtesis, D.: Run-time verification of behavioural conformance for conversational web services. In: 7th IEEE European Conference on Web Services (2009)
7. Dranidis, D., Metzger, A., Kourtesis, D.: Enabling Proactive Adaptation through Just-in-Time Testing of Conversational Services. In: Di Nitto, E., Yahyapour, R. (eds.) *ServiceWave 2010*. LNCS, vol. 6481, pp. 63–75. Springer, Heidelberg (2010)
8. Eilenberg, S.: *Automata, languages and machines*. Acad. Press, New York (1974)
9. Holcombe, M., Ipate, F.: *Correct Systems: Building Business Process Solutions*. Springer, Berlin (1998)
10. Holcombe, M., Coakley, S., Smallwood, R.: A general framework for agent-based modelling of complex systems. In: *European Conf. on Complex Systems* (2006)
11. Ipate, F.: Testing against a non-controllable stream x-machine using state counting. *Theoretical Computer Science* 353(1), 291–316 (2006)
12. Ipate, F., Holcombe, M.: An integration testing method that is proven to find all faults. *International Journal of Computer Mathematics* 63, 159–178 (1997)
13. Ipate, F., Holcombe, M.: Testing data processing-oriented systems from stream x-machine models. *Theoretical Computer Science* 403(2), 176–191 (2008)
14. Irvine, S., Pavlinic, T., Trigg, L., Cleary, J., Inglis, S., Utting, M.: Jumble java byte code to measure the effectiveness of unit tests. In: *Testing: Academic and Industrial Conference Practice and Research Techniques* (2007)
15. Kapeti, P., Kefalas, P.: A design language and tool for x-machines specification. In: *Advances in Informatics* (2000)
16. Kiran, M., Richmond, P., Holcombe, M., Chin, L.S., Worth, D., Greenough, C.: Flame: simulating large populations of agents on parallel hardware architectures. In: *9th International Conf. on Autonomous Agents and Multiagent Systems* (2010)
17. Ma, C., Wu, J., Zhang, T.: Sxmtool: A tool for stream x-machine testing. In: *World Congress on Software Engineering* (2010)
18. Ramollari, E.: Automated verification and testing of third-party Web services. Ph.D. thesis, University of Sheffield (2012)

A Low-Overhead, Value-Tracking Approach to Information Flow Security

Kostyantyn Vorobyov, Padmanabhan Krishnan, and Phil Stocks

Centre for Software Assurance
Bond University, Gold Coast, Australia
{kvorobyov,pkrishna,pstocks}@bond.edu.au

Abstract. We present a hybrid approach to information flow security where security violations are detected at execution time. We track secure values and secure locations at run time to prevent problems such as password disclosure in C programs. This analysis is safe in the presence of pointer aliasing. Such problems are hard to solve using static analysis (or lead to many false positives). Our technique works on programs with annotations that identify values and locations that need to be secure. We instrument the annotated program with statements that capture relevant information flow with assertions that detect any violation. This instrumentation does not interfere with the safe assignment of values to variables in the program. The instrumented assertions are invoked only when relevant values or locations are involved. We demonstrate the applicability of our approach by analysing various Linux utilities such as `su`, `sudo`, `passwd`, `ftp` and `ssh`. Our experiments show that for safe executions the overhead introduced by our instrumentation is, on average, less than 1%.

Keywords: Information flow, Program instrumentation, Assertion generation, Monitoring.

1 Introduction

The problem addressed by information flow security is to ensure that data identified as *secret* or *high security* are not exposed in any way external to the program under analysis. Values can be exposed by assignment to a *public* or *low security* variable, or by direct output, such as through a `print` function. Note, that the latter is essentially the same as exposed assignment as the value is assigned to the low security parameter of the function.

Traditional approaches to flow security [10,12] analyse programs with respect to program variables, which can involve a costly static analysis, and great imprecision when dealing with pointer aliasing. The core idea behind our approach is to dynamically track the *values* in a program, rather than the variables, to ensure that no secure value is leaked at run time. We present an approach to track secure values that uses a simple static analysis phase to instrument a program with assertions that enable dynamic monitoring of values and induce program

failure on a security violation. Our approach detects a wide range of security violations and works effectively for explicit flows in the presence of pointers. Our experiments indicate that the overheads introduced by our monitor are very low.

The flow of information in a program can be represented as a sequence of assignment statements: pairs $\langle addr, val \rangle$ such that a location in the memory $addr$ is assigned a value val . Note, the memory address is also a value. Such an assignment is safe if val is not secret or if memory location $addr$ can not be observed by an attacker. We refer to memory locations that can not be accessed by an attacker as *safe locations*. A safe program is one where all assignment statements are safe. Checking all assignments statically (at compile time) requires imprecise approximation, otherwise the complexity of such analysis is at least NP-hard and often undecidable.

Our approach enforces information flow safety during a dynamic run of a program (say P) by annotating it with a set of statements that detect vulnerabilities during execution and fail the program before a vulnerability occurs. An annotated program (say P') captures the set of secret values received by a program and verifies each assignment in P with respect to that set during its dynamic run. That is, before every assignment statement in P there is an assertion in P' which checks an assigned value and induces a failure if a vulnerability is detected.

```

1 function (int x)
2     char *high = "secret";
3     char *a = high;
4     char *low = "public";
5     if (x)
6         a = low;
7     low = strdup(a);
8 }
```

Listing 1. Vulnerabilities we detect

The C code fragment at Listing 1 demonstrates the types of issues we are able to solve using our approach. Let the character pointer `high` be a designated secure location and a value `secret` assigned to `high` be a secret value that should not be exposed to an attacker. Our analysis will verify the safety of assignment statements with respect to the secure value `secret` and a secure location `high`. The safe locations are the blocks of memory where secret values are stored: in this case addresses of memory blocks `high` points to (i.e., `&high[0]`). An aliasing statement at line 3 forces a character pointer `a` to point to a safe location. We verify this statement as safe, since no flow of data from the safe locations has occurred. We then identify the assignment at line 4 as safe (while `low` represents an unsafe location the value transferred to `low` is different to our secret value). As evaluation of values occurs at run time, we can correctly identify the safety of an assignment at line 7 during a run of a program. That is, if argument `x` is different to 0, then the statement at line 6 is executed, which implies that the assignment at line 7 is safe (a value assigned to an unsafe location is different to `secret`). This is because pointer `a` points to value `public`. If `x` evaluates to 0, the

statement at line 6 is not executed and we detect a vulnerability which transfers a secret value *secret* to an unsafe location pointed by *low*.

At this stage our approach does not detect a class of security violations caused by implicit information flows. For example, given the statement `if (high) low = 1; else low = 0;`, where *high* is safe, our approach fails to detect an unsafe implicit flow from *high* to *low*, which may disclose a branch taken during the execution. However we will detect all assignments of high security values to insecure locations.

The rest of the paper is organised as follows. Section 2 presents our technique at an abstract program level. Section 3 shows how to apply our technique for C programs. Section 4 discusses empirical results of a prototype implementation of our technique. Section 5 presents related work and Section 6 offers our conclusions.

2 Low-Overhead, Hybrid, Secure Value Tracking

Our approach is a hybrid static and dynamic technique operating on a program pre-annotated with locations of secure assignments marked. During the static stage we perform a number of source-to-source transformations on P to generate P' , which adds statements to track secure values and detect any insecure assignment of secure values. At the dynamic stage we only need to run P' . The assertions in P' observe the execution. A program passing the dynamic phase (no detected failures) does not leak any private information through insecure assignments.

Assignment is the only relevant operation to our analysis, so we consider programs at the abstract level of imperative sequences of assignments with minimal standard control structures and function calls, similar to how the program would look in Static Single Assignment form or as assembly code. We present our approach for an abstract imperative language, and show how it is implemented for C in Section 3. We first describe our memory model and the details of the imperative language enriched with pointers used to describe transformations. We then discuss the set of compositional transformation rules used to derive P' , and how the dynamic execution phase ensures safety.

2.1 Abstract Language

Figure 1 shows the syntax of our abstract imperative language. The set of variables $Vars$ is the set of elements generated by Var , which consists of primitive (indicated by $PrimVar$) and pointer types (indicated by the prefix **ptr**). Before we define the rest of the syntax, we describe our model of memory. Let $Vals$ be the set of values. A particular instance of the memory (or state) is represented as a function over the set of values (i.e., a function of the type $Vals \rightarrow Vals$). Every variable x from the set $Vars$ has a representation in memory. This is indicated by the function ρ where $\rho(x, m)$ represents the address of the variable x in memory m . The r-value of a variable can be obtained using the function `eval`, which

```

Var ::= PrimVar | ptr Var
e ::= x | e ⊕ e | f(e) | eval(x) | addressof(x)
c ::= skip | def(x) | x := e | if e then c1 else c2 | while e do c | c; c | assert(e)
f ::= Ident ≜ c
P ::=  $\tilde{f}$ ; e

```

Fig. 1. Abstract Language

can defined to be $m(\rho(x, m))$. For variables that are pointers, the address of the value that the pointer points to is given by $\text{addressof}(\text{ptr } x, m) = m(\rho(x, m))$ while the value that the pointer points to is $\text{eval}(\text{ptr } x, m) = m(m(\rho(x, m)))$. Primitive variables are direct mappings from addresses to their values in m , while **ptr** is a pointer dereferencing operator.

Expression e consists of variables x , composite expressions $e \oplus e$ (where \oplus is a binary operator), function calls $f(e)$, and operators $\text{eval}(x)$ and $\text{addressof}(x)$. Command c consists of atomic commands **skip**, assignment statements ($x := e$), sequential composition of commands ($c; c$), conditional expressions (**if** e **then** c_1 **else** c_2), loops (**while** e **do** c) and assertions (**assert**(e)). Command **assert**(e) terminates a program if an expression e evaluates to **false** and executes **skip** otherwise.

Function definitions consist of unique function names (*Ident*) followed by a command. A program P is a possibly empty sequence of function definitions followed by an expression.

We also assume the usual small step semantics [13] of executing a program p in state m . We denote this by $\langle p, m \rangle \mapsto \langle p', m' \rangle$ where the execution of a statement in p in state m yields the state m' and p' is the residual program. We define \mapsto to be reflexive transitive closure of \mapsto . We say a program terminates normally when p' is **skip**.

2.2 Static Program Instrumentation

The static phase of our analysis makes simple source-to-source transformations on a program. An input program P is such that it contains a number of annotated assignment statements. These annotations merely flag the assignments required to be safe. Such statements represent assignment of secret data received by a program to its variables. We treat memory locations that are assigned private values as safe. Annotated and non-annotated statements in P are distinguished using a boolean function $\Gamma(\text{stmt})$, which returns **true** if an assignment statement stmt in P is annotated and **false** otherwise.

The transformations instrument P with statements that track values during execution and assertions that will fail if security violations occur at run time. For the purposes of dynamic evaluation we define an auxiliary function $\text{compare}(val, V)$, which compares a value val to the elements of a collection V and evaluates to **true** if val is not found in V and to **false** otherwise. The **compare** function is left abstract at this stage since it depends on the implementation

$$\text{Define: } \frac{}{\mathbf{def}(x) \rightarrow \mathbf{def}(x)} \quad \text{Skip: } \frac{}{\mathbf{skip} \rightarrow \mathbf{skip}} \quad (1)$$

$$\text{Assignment1: } \frac{\Gamma(x := e) = \mathbf{true}}{x := e; \rightarrow x := e; \quad \begin{array}{l} H_{val} \text{ += eval}(x); \\ H_{var} \text{ += addressof}(x); \end{array}} \quad (2)$$

$$\text{Assignment2: } \frac{\Gamma(x := e) = \mathbf{false}}{x := e; \rightarrow \mathbf{def}(t); \quad \begin{array}{l} t := e; \\ \mathbf{if}(\mathbf{addressof}(x) \notin H_{var}) \\ \quad \mathbf{then} \mathbf{assert}(\mathbf{compare}(\mathbf{eval}(t), H_{val})); \\ \quad \mathbf{else} \mathbf{skip}; \\ x := t; \end{array}} \quad (3)$$

$$\text{Assert: } \frac{}{\mathbf{assert}(e) \rightarrow \mathbf{assert}(e)} \quad \text{If: } \frac{c_1 \rightarrow c'_1, c_2 \rightarrow c'_2}{\mathbf{if} e \mathbf{then} c_1 \mathbf{else} c_2 \rightarrow \mathbf{if} e \mathbf{then} c'_1 \mathbf{else} c'_2} \quad (4)$$

$$\text{While: } \frac{c \rightarrow c'}{\mathbf{while} e \mathbf{do} c \rightarrow \mathbf{while} e \mathbf{do} c'} \quad \text{Sequence: } \frac{c_1 \rightarrow c'_1, c_2 \rightarrow c'_2}{c_1; c_2 \rightarrow c'_1; c'_2} \quad (5)$$

$$\text{Function: } \frac{c \rightarrow c'}{f \triangleq c \rightarrow f \triangleq c'} \quad \text{Program: } \frac{\tilde{f} \rightarrow \tilde{f}'}{\tilde{f}; e \rightarrow \mathbf{def}(H_{var}); \mathbf{def}(H_{val}); \tilde{f}'; e} \quad (6)$$

Fig. 2. Rewrite Rules

language and the type of security analysis performed. We discuss constructing the `compare` function further in Section 3.

Figure 2 shows the full set of transformation rules applied on the input program P , which yields a modified program P' , equipped with statements and assertions required to prevent vulnerabilities caused by assignment of secret values to unsafe locations.

The first step of program instrumentation involves annotating an input program P with the set of statements that record secret values a program manipulates during a dynamic run. This step also identifies and records memory locations that are allowed to be assigned private data. Since our technique is based on value comparison, we identify safe locations using addresses of memory blocks of the actual data storage (retrieved using the `addressof` operation). To keep track of secure values and locations we use collections of values H_{var} and H_{val} , such that during the execution of P' , H_{var} will hold memory addresses of secure locations and H_{val} will hold secret values for a particular run. During the execution of P' , a memory block is considered to be safe (allowed to be assigned secret data), if its address is recorded in H_{var} . Similarly, assignment of a value v

to a non-secure location is considered safe if H_{val} does not contain v or another value that is similar to v (as determined by the `compare` function). Additionally, this step of program annotation involves explicit definitions of H_{var} and H_{val} (Figure 2, Rule *Program*).

The second step of the instrumentation inserts statements that record private values and secure locations. This is done for every annotated assignment statement in the input program P (i.e., those where $\Gamma(x := e)$ evaluates to `true`, Rule *Assignment1*). To record safe addresses we insert statements (immediately after annotated assignments) that retrieve addresses of declared safe memory blocks using the `addressof` operation. We add these addresses to the collection that tracks safe memory locations (H_{var}). This is followed by a statement that records a secret value (assigned to a safe location) to the collection that tracks secret values (H_{val}). Secret values are determined using the `eval` operation. Note, that application of `addressof` and `eval` operations guarantees that we correctly record addresses and values of memory blocks that contain data, not the addresses or values of the pointers that may point to them.

Finally, we enforce the safety of assignments that are not annotated in P (i.e., those where Γ evaluates to `false`, Rule *Assignment2*). For each such statement we insert a security assertion that verifies it with respect to H_{var} and H_{val} . Such an assignment statement is safe if either the address of the memory block that is being assigned a value is recorded in H_{var} (safe location) or if the value that flows to an unsafe memory location is different to any of the secure values stored in H_{val} . We introduce the temporary variable t so that we do not evaluate e twice. The value returned by e is necessary to determine if it is a secure value. Furthermore, if safe, we also wish to assign this value to x . Note, this use of assertions is a monitoring approach as a program failure is induced before a secret value flows to an unsafe location.

We consider `skip` and `def` statements to be safe, because they do not assign data and thus can not jeopardise the safety of a run. These statements are left unchanged. For any other statements (such as conditionals and loops) we recursively apply previous rules.

2.3 Execution of Instrumented Program

During the dynamic stage, the instrumented program P' monitors information flow safety of an original program (P). A failure of a security assertion is a prevented security vulnerability. On execution, H_{var} and H_{val} are initialised to empty collections. As execution proceeds, values and addresses for annotated assignments are added to H_{var} and H_{val} . Note, that before H_{var} and H_{val} are appended with values any assignment statement is safe, since there is no designated private data available for comparison. Non-annotated assignment statements are evaluated with respect to data stored in H_{var} and H_{val} . If an unsafe location is being assigned a value, P' invokes `compare` which determines the safety of the value with respect to the values stored in H_{val} . For the cases when `compare` determines similarity between assigned values and data from H_{val} , execution is aborted and the vulnerability that occurred is reported. A program

execution passing the dynamic analysis phase (no detected failures) does not leak any secure information (via assignment of data) for that particular run.

Our framework can be formally described as follows. Let p be a program and X_s be the set of secure variables and V_s be the set of secure values obtained from the annotations.

Definition 1. An execution of p in state m is safe if and only if $\langle p, m \rangle \mapsto \langle p', m' \rangle$ and for every location loc and for every variable x belonging to X_s , with $\rho(x, m')$ not equal to loc implies $m'(loc)$ does not belong to V_s .

This can also be stated as:

$$\langle p, m \rangle \mapsto \langle p', m' \rangle \Rightarrow \{m'(loc) | loc \in \text{dom}(m') \setminus \{\rho(x, m') | x \in X_s\}\} \cap V_s = \emptyset.$$

Theorem 1. Let $p \rightarrow q$ using the rewrite rules shown in Figure 2.

If every execution of p in state m is safe then every execution of q in m is safe. Moreover if the execution of p terminates normally, the execution of q will also terminate normally.

If there is an execution of p in state m that is unsafe, then the execution of q in state m will be safe but q will not terminate normally.

The above theorem can be proven via induction over the structure of the program. The proof itself is standard but tedious. It relies on H_{val} having all values that are comparable to values in V_s and H_{var} having the locations corresponding to X_s . \square

3 Application to C programs

The rewrite rules discussed in Section 2 are expressed on an abstract language and need to be mapped to a concrete level to apply to a real implementation language. Further, implementations of the `compare` function, `addressof` function, and collections H_{var} and H_{val} need to be provided. Note, this means that `compare` can be tailored to suit specific flow safety definitions/requirements. This section discusses how to adopt our approach for programs written in the C programming language.

H_{var} and H_{val} . To store safe locations and values of variables we use globally defined arrays. H_{var} is an array of integers (i.e., `intptr_t`), which stores safe addresses. As C is a typed language, we implement H_{val} as a collection of arrays per data type. These arrays can grow dynamically as desired.

Memory Addresses. Since in C variables may span across multiple memory blocks, we represent memory locations that identify secure or insecure locations as start addresses of consecutive memory blocks that hold actual data. For instance, if a character pointer (say `char *ptr`) is secure we use the location `&ptr[0]`. For composite data types (`struct` or `union`) we use the collection of the start location of the structure and the start locations of each field. For example,

`struct stt {char *str, int i}` `st` has the memory locations `{ &st, &st.str[0], &st.i }`.

The `offsetof` operation is implemented as a set of functions. For each data type that we handle (say `T`) we define `intptr_t offsetof_T(T var)`, which extracts addresses for variables of type `T`. For example, for type `char*` we define `intptr_t offsetof_charPtr (char* var)`, which returns `&var[0]`. Note, that this can be determined statically.

Library or External Functions. Code in function bodies that are available is instrumented as described above. However, approximations are required for library or external functions for which the source code is unavailable. Calls to external functions are potential security flow hazards because function calls may leak their arguments. For example, `printf("%s", ptr)` is a security violation if `&ptr[0]` is a safe location. Functions for which source code is unavailable are annotated before analysis as either *safe* or *unsafe*. Every call to an unsafe external function is instrumented with an assertion that evaluates the function arguments and fails if any of the arguments refers to a safe location or value. For example, a call to the unsafe standard function `printf("%s", ptr)` is transformed to `if (eval(ptr) ∈ Hvar) assert(0); printf("%s", ptr);`.

External function calls also may potentially make assignments to pointer arguments. For safety, we over-approximate that any parameter of an external function that is a mutable pointer results in an assignment of data. The program is analysed as if there were an assignment of that parameter's value to itself immediately after the execution of the function. For example, for a function call `strcpy(d,s)` to library function `strcpy(char *dest, const char *src)` we assume `d` is assigned in the body of `strcpy` and treat the call as if it were `strcpy(d,s); d = d;`. No assertions are generated for arguments that are constant arguments as these cannot be changed.

Value Comparison. The `compare` function can be simply implemented as a direct look-up based on equality. However, more sophisticated or tailored comparisons may also be used. Our initial studies primarily target password flow analysis, so we implement `compare` as a function that uses Levenshtein distance [5] as the measure of similarity between values (strings). Levenshtein distance is frequently used to evaluate the strength of passwords against dictionaries [6]. It is computed using the number of edits required to transform one string into another. Our case studies implement `compare` as failing if the Levenshtein distance between strings is less than some pre-defined threshold and succeeding otherwise. The benefit of using this measure is that we can detect similar, but not identical strings, for example, partially exposed passwords. However, this implementation may induce failures for different, but very short strings. For instance, `compare` will return 0 when comparing any strings of length less than the threshold.

Safe Termination. C memory deallocation procedures (e.g., `free`) do not guarantee destruction of values. This may result in disclosure of private values left

in memory after a program terminates. Our analysis checks that a program correctly cleans up its private values by first disabling memory deallocation functions and then, before program termination, checking that no location in H_{var} contains any value in H_{val} . This is easily done using C standard library functions `atexit`, `on_exit` and `signal`. Note that while this approach is adequate for our experimentation purposes, a more sophisticated approach that remembers designated safe memory addresses and scans possibly freed memory on termination could be used in production monitoring systems.

3.1 Example of Instrumentation

Listing 2 shows the resulting instrumented program for the program in Listing 1 where the assignment at line 2 is annotated to be safe. To simplify the presentation we omit checks for safe termination procedures.

```

1 char *hval[]; // -- Values that we track intptr_t hvar[]; // --
2 Safe addresses
3
4 // Compare function: return addressof(var) ∈ Hvar or eval(var) ∉ Hval
5 int compare (char *var, char *hval[], intptr_t hvar[]);
6
7 // Track values: Hvar += addressof(var), Hval += eval(var)
8 void record_safe(char *var, intptr_t hvar[], char *hval[]);
9
10 // Instrumented program
11 function (int x)
12     char *high = "secret";
13     /* Instrumented */ record_safe(high, hvar, hval);
14
15     /* Instrumented */ assert (compare(a, hvar, hval));
16     char *a = high;
17
18     /* Instrumented */ assert (compare(low, hvar, hval));
19     char *low = "public";
20
21     if (x) {
22         /* Instrumented */ assert (compare(a, hvar, hval));
23         a = low;
24     }
25     /* Instrumented */ assert (compare(a, hvar, hval));
26     low = strdup(a);
27 }

```

Listing 2. Instrumented program P'

We instrument the program as follows. We first add global definitions of structures to hold the addresses and values that we track (`hval` and `hvar` arrays in lines 1 and 2 respectively). We then provide definitions of functions `compare` and `record_safe` (in lines 5 and 8 respectively) required for analysis. We do not show the body of these functions here. The function `compare` checks the address of the provided pointer and succeeds if it is a safe location (e.g. `&var[0]` is stored in `hvar`). The function `record_safe` records addresses and values of safe locations to `hvar` and `hval`. The call to `record_safe` at line 13 records `&high[0]` to `hvar` and "secret" to `hval`. Further, for every assignment that is not annotated (and hence

could potentially involve a low security location), we add an assertion statement (lines 15, 18, 22 and 25) which checks the type of address that is being assigned a value (high or low) and an assigned value.

4 Implementation and Empirical Results

We have implemented our technique in a prototype tool built on top of the *Clang* [1] compiler architecture (LLVM project, version 2.9 [3]), for programs written in the C programming language. The platform for all results reported here was Intel Core i5-2400 3.1 GHz machine with 4GB of RAM, running Gentoo Linux.

To evaluate our approach we have checked the safety of password flow in six UNIX utilities using our prototype: `su`, `sudo`, `passwd`, `vlock`, `ftp`, and `dropbear` which is an implementation of `ssh`. The choice of utilities was dictated by the availability of password-authenticating functionality at a system level—tools for which safe password flow is important. These utilities differ in both functionality and implementation. All these tools are mature and have been extensively tested, yet we have detected leaks in `su` and `ftp`: `su` does not safely destroy the hash sum of the plain-text password and `ftp` does not overwrite a pointer where the plain-text password value received from the user is stored.

Table 1. Run time statistics

| Program | LOC | Overall Time (s) | Annotations | Assertions | Recompile Time (s) |
|------------------------------|-------|------------------|-------------|------------|--------------------|
| <i>su</i> (coreutils 8.13) | 2342 | 1.277 | 3 | 3 | 0.261 |
| <i>sudo</i> (1.8.2) | 5266 | 1.944 | 1 | 14 | 2.327 |
| <i>passwd</i> (shadow 4.1.4) | 3465 | 2.340 | 3 | 64 | 0.289 |
| <i>vlock</i> (2.2) | 1635 | 0.296 | 1 | 12 | 0.162 |
| <i>ftp</i> (inetutils 1.8) | 9595 | 2.791 | 2 | 99 | 1.206 |
| <i>dropbear</i> (2011.54) | 14533 | 9.415 | 1 | 5 | 2.146 |

Table 1 shows instrumentation statistics of our prototype. The overall time measures time for applying our technique on the source programs, and recompile time measures the time taken to compile the instrumented program. Table 1 also shows the number of injected assertions and the number of annotations, which flag the assignment of password data to program variables¹.

It can be seen that the speed of our transformation directly depends on the complexity of the source code, ranging from 0.296 seconds for `vlock` (1635 lines) to approximately 9.5 seconds for `dropbear` (14533 lines). Generating the assertions does not take more than 0.01 seconds. Most of the time is spent on parsing and generating abstract syntax trees. The average parsing time per tool is approximately 70% of the overall time and ranges from the maximum of 81% for

¹ manually located and identified.

`ftp` to the minimum of 51% for `vlock`. The rest of the time is mostly spent on the output of the modified files.

The time required to recompile generated source code directly depends on the number source files that are modified and the dependencies that need to be preserved in the build. In the majority of the cases this time is smaller than the analysis time. For example in `su` only 1 file needs to be recompiled, hence the minimal compile time of 0.261 seconds. In some cases, however, for example for `sudo`, the recompile time appears to be greater than the analysis time due to dependencies in source files. Note, neither recompile nor analysis time depends directly on the number of generated assertions, but on the size of the software.

These results suggest that our technique may scale well for large and complex software, because analysis time is dependent on the size of the software, rather than on the complexity of the constructs, since our transformations are purely syntactic. Also this entire instrumentation process is performed only once.

Figure 3 shows run time overhead statistics for programs instrumented by our technique. To reliably determine overheads produced by our technique we performed 50 series of runs of modified and original executables, such that each series contains 1000 runs. This is to account for variance produced by the automation (`expect` framework), networking (for `ftp` and `dropbear`) and system I/O. Also the execution time of a single run of programs such as `su`, `vlock`, and `sudo` is too small to measure accurately. We add a loop to execute each program 1000 times to get a single measurement.

Additionally, to correctly calculate the overheads, we disabled `abort` statements in instrumented assertions. Assertions are evaluated and violations reported, but program failure is not triggered. This is due to the discovered vulnerabilities in `su` and `ftp` utilities, which result in failures of instrumented programs regardless of the input.

Figure 3 shows the running times of original and instrumented programs. The figure adjacent to the name of the program is the percent overhead. The bars indicate standard deviation. The times reported refer to average running time per 1000 runs, per 50 series.

In continuous runs of the modified and original versions of the tools we have used correct password values of 10 characters in length. This is to avoid failures due to short strings comparison and ensure that the run of a modified program invokes the majority of the instrumented assertions. We calculate the overhead for realistic password values.

It can be seen that overheads produced by the application of our technique do not exceed 0.5% and range from 0.06% for `ftp` to 0.049% for `passwd`. Standard deviation for the the majority of the tools is also small, which allows us to conclude that overhead results are precise. For `dropbear`, however, the standard deviation is considerably greater for the instrumented program than for the original. This is because of variable response times of the network, since during experimentation `dropbear` was configured to connect and disconnect to a real SSH server. Despite the influence of the network, the variance is as small as 0.05 seconds per run.

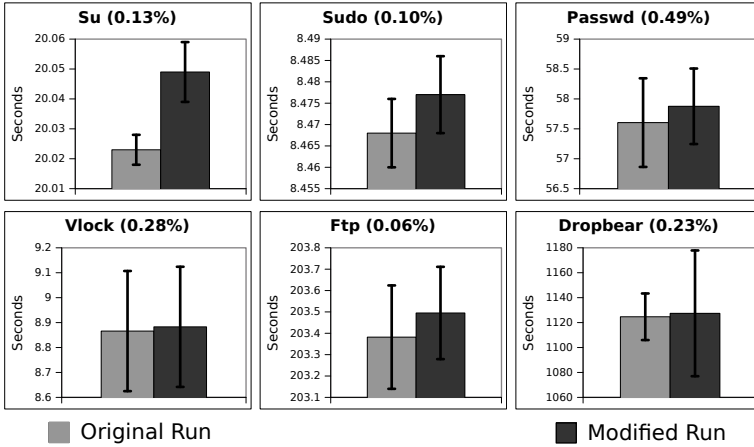


Fig. 3. Results of Experiments

It can be seen that the number of generated assertions does not impact overhead produced by the instrumentation. For example, overhead produced for `ftp` (0.06%) is the lowest despite having the largest number of injected assertions (99). This is because the main factor that effects overhead is the length of the strings that need to be compared and the number of secure values tracked. For instance, in `passwd` a password is entered three times (old once, new twice), hence the largest overhead of 0.49%.

We now discuss issues that could affect the validity of our results. The first is the choice of programs in our experiments. Although these are realistic tools we cannot claim they are representative of all programs in the security domain. The second issue is the number of values and locations we track. We believe we track all values and locations relevant to security. But, since the annotation process is manual, some values or locations may potentially have been overlooked. This could result in lower overheads. The final issue is related to the input values chosen for the execution of the systems. While we have chosen typical values of passwords, there is no guarantee that these passwords exercised all paths within the program. We have not implemented any coverage metric. Thus there could be paths that might lead to higher overheads. While we do not report any security violations for some of the programs (e.g, `dropbear` and `vlock`) these programs may have information flow vulnerabilities. Of course, our work is not about certifying programs – thus we can claim that all the paths that we have tested for these programs are safe.

5 Related Work

Security analysis using information flow is a well studied area. Here we summarise only those papers that are directly relevant to our approach.

Language-based information flow analysis [10,16] adds annotations to programs that enable information analysis. However adding such annotations is non-trivial especially given the semantics of the annotations. The annotations required in our approach are only to identify secure locations and values.

Russo and Sabelfeld [9] describe the trade-off between static and dynamic information flow analysis. They present a framework for hybrid analysis. While the paper’s main contribution is expressivity and impossibility results, its framework enables us to classify our work as a hybrid (program transformation occurs statically and value tracking occurs at run time) monitoring technique. Our program annotations assign security levels to memory locations and these security levels do not change. However, at the programming language level, because of aliasing, the security level associated with a variable, especially of a pointer, can change dynamically.

There are a number of theoretical approaches to dynamic information flow analysis. For instance, Shroff et al. [11] present a theoretical framework for monitoring based information flow. Their framework detects both direct and indirect information flows and thus implements dynamic non-interference. They show that the run time overhead is $\mathcal{O}(n^2)$ where n is the number of program points. Austin and Flanagan [2] present two semantics for information flow. The first semantics (called universal labelling) dynamically labels each value while the second semantics (called sparse labelling) labels only values that flow from one domain to another. The theory is described using a variant of the λ -calculus. They have built an interpreter for their system and show that sparse labelling is much better than universal labelling, but the overheads even for sparse labelling can be close to 100%. It is not clear how these results translate to real programs.

We now review practical approaches to information flow analysis. Le Guernic et al. [4] analyse information flow in both sequential and concurrent programs. They use static analysis at run time and to detect indirect flows, which has the usual issues with static analysis such as false positives. Furthermore, they do not handle pointers.

LIFT [8] tracks values at the binary level. This is achieved by tagging the values. The overhead for propagating the tags and checking the tags can be quite high. It varies from about 6% to 300%. Their main advantage is the ability to track information flow across library calls.

Panorama [11] performs security information flow analysis in three stages – testing, monitoring and analysing. First, the code under investigation is loaded into the testing environment, where the set of automated tests are conducted and program behaviour is monitored. The result obtained from monitoring is then analysed with respect to user-defined security requirements. The focus in Panorama is to track information flow under test cases.

LeakProber [15] provides profiling of paths that leak data. Their analysis is based on computing data flow graphs and tracking the appropriate path at run time. Their aim is to identify vulnerabilities by comparing normal and insecure data propagation graphs. They also focus mainly on data that crosses the

user/kernel boundary. For that they patch and recompile the kernel to support profiling.

Resin [14] is similar to our work and tracks values and has assertions that are checked at run time. Web-based systems is the main application domain and Resin supports the specification of policies and filters especially when data leaves one domain. As they mainly support programs written in PHP and Python and track only secure values and not secure locations, they do not need to handle aliasing. A significant limitation of their approach is the need to modify the interpreter to handle the security policies. Our work modifies only the input program and standard tools (such as `gcc`) can still be used. Resin has high overheads (more than 400% for some SQL related operations) while our overheads are negligible.

Magazinius et al. [7] inject monitors at the source code level. These monitors are similar to the assertions we insert into the programs. They handle code that is evaluated on the fly (i.e., executing strings as code), but they do not handle pointers (or aliasing) and also assume that functions have no side-effects. The overheads of their approach are also high, ranging from 20% to 1700%. In C programs, executing strings as code is not an issue.

In summary, we focus on programs written in C where issues such as aliasing make static analysis difficult or unreliable. Although we do not handle implicit flows, the overheads of our approach are much lower than any of the other approaches.

6 Conclusions

In this paper we have described a hybrid approach for information flow that tracks secure values and locations. We assume that we have the source code of the program whose execution we wish to monitor for information leakage. We instrument the input program with statements to track relevant information flows. These statements record secure values (using H_{val}) and locations (using H_{var}) that are involved in the flow. The use of H_{var} enables us to handle pointer aliasing. We also insert assertions that determine if the high security values flow into low security locations. This is achieved using the function `compare` which checks that a location that is not in H_{var} does not receive a value in H_{val} . The exact values in H_{var} and H_{val} are calculated at execution time for the particular run of the program. The inserted assertions are executed before the actual assignment occurs. Hence, we have developed a monitoring approach for secure information flows. We have implemented the approach for C programs using *Clang* compiler. Our experiments with various Linux programs show that the overhead to detect the leakage of passwords is less than 1%. This is because H_{val} and H_{var} are implemented as arrays. Updating these arrays at run time involves simple assignment statements. It is the execution of `compare` that is most time consuming. Finally, our approach has no false positives. A program run that does not violate any of the inserted assertions will not leak any secure information (via assignment).

Acknowledgement. Kostyantyn Vorobyov is supported by a grant from Oracle Labs.

References

1. clang: a C language family frontend for LLVM (March 2012), <http://clang.llvm.org>
2. Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. In: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security, PLAS 2009, pp. 113–124. ACM (2009)
3. Lattner, C., Adev, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization, Palo Alto, California (March 2004)
4. Le Guernic, G., Banerjee, A., Jensen, T.P., Schmidt, D.A.: Automata-Based Confidentiality Monitoring. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 75–89. Springer, Heidelberg (2008)
5. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions and reversals. Soviet Physics Doklady 10(8), 707–710 (1966)
6. Ma, W., Campbell, J., Tran, D., Kleeman, D.: Password entropy and password quality. In: International Conference on Network and System Security, pp. 583–587 (2010)
7. Magazinius, J., Russo, A., Sabelfeld, A.: On-the-fly Inlining of Dynamic Security Monitors. In: Rannenber, K., Varadharajan, V., Weber, C. (eds.) SEC 2010. IFIP AICT, vol. 330, pp. 173–186. Springer, Heidelberg (2010)
8. Qin, F., Wang, C., Li, Z., Kim, H., Zhou, Y., Wu, Y.: Lift: A low-overhead practical information flow tracking system for detecting security attacks. In: 39th Annual IEEE/ACM International Symposium on Microarchitecture, Orlando, Florida, USA, pp. 135–148 (December 2006)
9. Russo, A., Sabelfeld, A.: Dynamic vs. static flow-sensitive security analysis. In: Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, pp. 186–199. IEEE (2010)
10. Sabelfeld, A., Myers, A.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (2003)
11. Shroff, P., Smith, S.F., Thober, M.: Dynamic dependency monitoring to secure information flow. In: Proceedings of the 20th IEEE Computer Security Foundations Symposium, CSF 2007, pp. 203–217. IEEE (2007)
12. Smith, G.: Principles of secure information flow analysis. In: Christodorescu, M., Jha, S., Maughan, D., Song, D., Wang, C. (eds.) Malware Detection, Advances in Information Security, vol. 27, pp. 291–307. Springer US (2007)
13. Winskel, G.: The formal semantics of programming languages - an introduction. Foundation of computing series. MIT Press (1993)
14. Yip, A., Wang, X., Zeldovich, N., Kaashoek, M.F.: Improving application security with data flow assertions. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP 2009, pp. 291–304. ACM (2009)
15. Yu, J., Zhang, S., Liu, P., Li, Z.: Leakprober: a framework for profiling sensitive data leakage paths. In: Proceedings of the 1st ACM Conference on Data and Application Security and Privacy, CODASPY 2011, pp. 75–84. ACM, New York (2011)
16. Zheng, L., Myers, A.C.: Dynamic security labels and static information flow control. International Journal of Information Security 6(2-3), 67–84 (2007)

Author Index

- Abdelhalim, Islam 248
Ait Mohamed, Otmane 263
Amighi, Afshin 33
Ayrault, Philippe 278
- Barnat, Jiří 48
Bauch, Petr 48
Benayoun, Vincent 278
Bicknell, Brett 323
Biechele, Florian 203
Bistarelli, Stefano 308
Bratanis, Konstantinos 352
Brim, Luboš 48
Butler, Michael 78, 323
- Ciobanu, Gabriel 141
Colley, John 323
Colombo, Christian 218
Conserva Filho, Madiel 342
Cuoq, Pascal 233
- Debbabi, Mourad 263
de C. Gomes, Pedro 33
Di Cosmo, Roberto 156
Din, Crystal Chang 94
Dovland, Johan 94
Dranidis, Dimitris 352
Dubois, Catherine 278
- Ferrara, Pietro 63
Francalanza, Adrian 218
Fuchs, Raphael 63
- Gurov, Dilian 33
- He, Jifeng 126
Huisman, Marieke 33
- Ipate, Florentin 293, 352
- Januzaj, Visar 203
Jaskolka, Jason 109
Jones, Cliff B. 1
Juhasz, Uri 63
- Khedri, Ridha 109
Kirchner, Florent 233
Kosmatov, Nikolai 233
Koutny, Maciej 141
Krishnan, Padmanabhan 367
Kugele, Stefan 203
- Lefticaru, Raluca 293
- Ma, Chris 172
Matichuk, Daniel 333
Mauersberger, Ralf 203
Mizzi, Ruth 218
Murray, Toby 333
- Nikolić, Đurica 16
- Oliveira, Marcel Vinicius Medeiros 342
Ouchani, Samir 263
Owe, Olaf 94
- Pace, Gordon J. 218
Paul, Wolfgang 188
Pessaux, François 278
Prevosto, Virgile 233
Priami, Corrado 16
- Qin, Shengchao 172
Qiu, Zongyan 172
- Reis, Jose 323
Rezazadeh, Abdolbaghi 78
- Salehi Fathabadi, Asieh 78
Santini, Francesco 308
Schmaltz, Sabine 188
Schneider, Steve 248
Shadrin, Andrey 188
Signoles, Julien 233
Snook, Colin 323
Steggles, Jason 141
Stocks, Phil 367
- Treharne, Helen 248
- Vorobyov, Kostyantyn 367

Xu, Qiwen 172

Yakobowski, Boris 233

Zacchiroli, Stefano 156

Zavattaro, Gianluigi 156

Zhang, Qinglei 109

Zhao, Yongxin 126

Zhu, Huibiao 126, 172

Zhu, Longfei 126

Zunino, Roberto 16