Einar Broch Johnsen
Luigia Petre (Eds.)

LNCS 7940

Integrated Formal Methods

10th International Conference, IFM 2013
Turku, Finland, June 2013
Proceedings

Springer

Lecture Notes in Computer Science 7940

Einar Broch Johnsen Luigia Petre (Eds.)

Integrated Formal Methods

10th International Conference, IFM 2013
Turku, Finland, June 10-14, 2013
Proceedings

Springer

Volume Editors

Einar Broch Johnsen
University of Oslo, Department of Informatics
P.O. Box 1080, 0316 Oslo, Norway
E-mail: einarj@ifi.uio.no

Luigia Petre
Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5A, 20520 Turku, Finland
E-mail: lpetre@abo.fi

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-38612-1 e-ISBN 978-3-642-38613-8
DOI 10.1007/978-3-642-38613-8
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013938948

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

LNCS Sublibrary: SL 2 – Programming and Software Engineering

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

Formal methods allow the modeling and analysis of various aspects of a complex system. Modeling languages differ in the system aspects they target, for which models can be naturally and succinctly developed. Numerous techniques address model analysis in these languages, specialized for different kinds of properties. Applying formal methods to complex systems often involves combining several models in different languages and exploiting the strengths of many analysis techniques. The integrated Formal Methods (iFM) conference series seeks to further research into hybrid approaches to formal modeling and analysis, and into the combination of (formal and semi-formal) modeling and analysis methods in all aspects of software development from language design through verification and analysis techniques to tools and their integration into software engineering practice. This volume includes the articles presented at the 10th edition of iFM.

The 10th International Conference on integrated Formal Methods (iFM 2013) was held during June 12-14, 2013, in Turku, Finland. The conference was organized by the Department of Information Technologies at Åbo Akademi University. Previous editions of iFM were held in York, UK (1999), Schloss Dagstuhl, Germany (2000), Turku, Finland (2002), Kent, UK (2004), Eindhoven, The Netherlands (2005), Oxford, UK (2007), Düsseldorf, Germany (2009), Nancy, France (2010), and Pisa, Italy (2012).

The conference has grown tremendously in the past years. iFM 2013 received 106 abstracts and 84 full paper submissions. The Program Committee ensured that each paper received three reviews and was carefully discussed, before selecting 25 papers for presentation at the conference. This leads to an acceptance rate of almost 30%. The scientific program of iFM 2013 was further strengthened by four outstanding invited speakers:

- Jean-Raymond Abrial, Marseille, France: *From Z to B and then Event-B: Assigning Proofs to Meaningful Programs*
- Susanne Graf, VERIMAG, France: *Knowledge for the Distributed Implementation of Constrained Systems*
- Kim Larsen, Aalborg University, Denmark: *Priced Timed Automata and Statistical Model Checking*
- Cosimo Laneve, University of Bologna, Italy: *An Algebraic Theory for Web Service Contracts*

The invited speakers have contributed papers to the proceedings that survey their work in these areas.

iFM 2013 attracted broad international interest. The authors of the submitted papers were affiliated to 34 countries spread out on all five continents. The authors of the accepted papers were affiliated to 14 countries, from Europe, Asia, and South and North America. The Program Committee was also very international, its members being affiliated to 16 countries, from Europe, North America, Asia, and Australia. The biggest number of accepted authors came from France and the biggest number of PC members came from the UK.

Associated with iFM 2013, the following workshops and tutorials were organized during June 10–11, 2013:

- The 4th International Workshop on Computational Models for Cell Processes
- Rodin User and Developer Workshop 2013
- BCS FACS 2013 Refinement Workshop 2013
- Tutorial on the Specification and Proof of Programs with Frama-C

These events significantly contributed to an exciting scientific program during an entire week.

To our great sadness, Professor Kaisa Sere from Åbo Akademi University passed away in December 2012. Kaisa was a renowned researcher in formal methods and one of the PC chairs of iFM 2002. She was happy that Åbo Akademi University were planning was planning to host the conference again in 2013. We are very grateful that one of her close scientific collaborators, Emil Sekerinski, McMaster University, Canada, has accepted to give a short talk at iFM 2013 on Kaisa's achievements in computer science.

We warmly thank the Program Committee of iFM 2013 for their excellent work, their high-quality reviews, their timeliness and enthusiasm, as well as for their determination to only accept the best papers with respect to novelty, innovation, and technical merit. It was an honor and a pleasure to work with you! We would also like to acknowledge and thank the reviewers that supported the Program Committee. The work of the Program Committee was supported from the beginning by the EasyChair software: we thank Andrei Voronkov for making this framework available. We are deeply indebted to the sponsors of iFM 2013: their generous support enabled a pleasant environment and nice social events, truly contributing to community building.

In the end, it is the authors of the contributed papers that made iFM 2013 a reality and a success. Thank you very much for your dedication: it is your work that makes up these proceedings!

April 2013

Einar Broch Johnsen
Luigia Petre

Organization

Program Committee

Erika Abraham	RWTH Aachen University, Germany
Elvira Albert	Complutense University of Madrid, Spain
Marcello Bonsangue	Leiden University, The Netherlands
Phillip J. Brooke	Teesside University, UK
Ana Cavalcanti	University of York, UK
Dave Clarke	Catholic University of Leuven, Belgium
John Derrick	Unversity of Sheffield, UK
Jin Song Dong	National University of Singapore, Singapore
Kerstin Eder	University of Bristol, UK
John Fitzgerald	Newcastle University, UK
Andy Galloway	University of York, UK
Marieke Huisman	University of Twente, The Netherlands
Reiner Hähnle	Technical University of Darmstadt, Germany
Einar Broch Johnsen	University of Oslo, Norway
Peter Gorm Larsen	Aarhus University, Denmark
Diego Latella	ISTI-CNR, Pisa, Italy
Michael Leuschel	University of Düsseldorf, Germany
Shaoying Liu	Hosei University, Japan
Michele Loreti	Università degli Studi di Firenze, Italy
Dominique Mery	Université de Lorraine, LORIA, France
Stephan Merz	INRIA Lorraine, France
Richard Paige	University of York, UK
Luigia Petre	Åbo Akademi University, Finland
Kristin Yvonne Rozier	NASA Ames Research Center, USA
Philipp Ruemmer	Uppsala University, Sweden
Thomas Santen	European Microsoft Innovation Center, Germany
Ina Schaefer	Technische Universität Braunschweig, Germany
Steve Schneider	University of Surrey, UK
Emil Sekerinski	McMaster University, Canada
Graeme Smith	University of Queensland, Australia
Colin Snook	University of Southampton, UK
Kenji Taguchi	AIST, Japan
Helen Treharne	University of Surrey, UK
Heike Wehrheim	University of Paderborn, Germany
Herbert Wiklicky	Imperial College London, UK
Gianluigi Zavattaro	University of Bologna, Italy

Additional Reviewers

Organization and Sponsors

The Organizing Committee of iFM 2013 consisted of Luigia Petre (chair), Maryam Kamali, Yuliya Prokhorova, Magnus Dahlvik, Nina Rytkönen, Tove Österroos and Susanne Ramstedt. The Workshops and Tutorials Chair was Pontus Boström.

We are very grateful to the financial and administrative support of the following institutions:

Department of Information Technologies at Åbo Akademi University, Springer, the Foundation for Åbo Akademi University (Stiftelsen för Åbo Akademi), and the Federation of Finnish Learned Societies (Tieteellisten seurain valtuuskunta, TSV). Their logos are gratefully displayed below:

Table of Contents

Invited Paper 1

From Z to B and then Event-B: Assigning Proofs to Meaningful Programs 1
Jean-Raymond Abrial

Refinement, Integration, Translation

Systems Design Guided by Progress Concerns 16
Simon Hudon and Thai Son Hoang

Assume-Guarantee Specifications of State Transition Diagrams for Behavioral Refinement 31
Christian Prehofer

Translating VDM to Alloy 46
Kenneth Lausdahl

Verification of EB^3 Specifications Using CADP 61
Dimitris Vekris, Frédéric Lang, Catalin Dima, and Radu Mateescu

Invited Paper 2

Knowledge for the Distributed Implementation of Constrained Systems (Extended Abstract) 77
Susanne Graf and Sophie Quinton

Verification

Automated Anonymity Verification of the ThreeBallot Voting System 94
Murat Moran, James Heather, and Steve Schneider

Compositional Verification of Software Product Lines 109
Jean-Vivien Millo, S. Ramesh, Krishna Shankara Narayanan, and Ganesh Khandu Narwane

Deductive Verification of State-Space Algorithms 124
Frédéric Gava, Jean Fortin, and Michael Guedj

Inductive Verification of Hybrid Automata with Strongest Postcondition Calculus 139
Daisuke Ishii, Guillaume Melquiond, and Shin Nakajima

Invited Paper 3

Priced Timed Automata and Statistical Model Checking 154
Kim Guldstrand Larsen

Reachability and Model Checking

Improved Reachability Analysis in DTMC via Divide and Conquer 162
Songzheng Song, Lin Gui, Jun Sun, Yang Liu, and Jin Song Dong

Solving Games Using Incremental Induction 177
Andreas Morgenstern, Manuel Gesell, and Klaus Schneider

Model-Checking Software Library API Usage Rules 192
Fu Song and Tayssir Touili

Formal Modelling and Verification of Population Protocols 208
Dominique Méry and Michael Poppleton

Usability and Testing

Detecting Vulnerabilities in Java-Card Bytecode Verifiers Using Model-Based Testing 223
Aymerick Savary, Marc Frappier, and Jean-Louis Lanet

Integrating Formal Predictions of Interactive System Behaviour with User Evaluation 238
Rimvydas Rukšėnas, Paul Curzon, and Michael D. Harrison

Automatic Inference of Erlang Module Behaviour 253
Ramsay Taylor, Kirill Bogdanov, and John Derrick

Distributed Systems

Integrating Proved State-Based Models for Constructing Correct Distributed Algorithms 268
Manamiary Bruno Andriamiarina, Dominique Méry, and Neeraj Kumar Singh

Quantified Abstractions of Distributed Systems 285
Elvira Albert, Jesús Correas, Germán Puebla, and Guillermo Román-Díez

Invited Paper 4

An Algebraic Theory for Web Service Contracts 301
Cosimo Laneve and Luca Padovani

Semantics

A Compositional Automata-Based Semantics for Property Patterns 316
Kalou Cabrera Castillos, Frédéric Dadeau, Jacques Julliand, Bilal Kanso, and Safouan Taha

A Formal Semantics for Complete UML State Machines with Communications 331
Shuang Liu, Yang Liu, Étienne André, Christine Choppy, Jun Sun, Bimlesh Wadhwa, and Jin Song Dong

From Small-Step Semantics to Big-Step Semantics, Automatically 347
Ştefan Ciobâcă

Program Equivalence by Circular Reasoning 362
Dorel Lucanu and Vlad Rusu

System-Level Analysis

Structural Transformations for Data-Enriched Real-Time Systems 378
Ernst-Rüdiger Olderog and Mani Swaminathan

Deadlock Analysis of Concurrent Objects: Theory and Practice 394
Elena Giachino, Carlo A. Grazia, Cosimo Laneve, Michael Lienhardt, and Peter Y.H. Wong

Broadcast, Denial-of-Service, and Secure Communication 412
Roberto Vigo, Flemming Nielson, and Hanne Riis Nielson

Characterizing Fault-Tolerant Systems by Means of Simulation Relations 428
Ramiro Demasi, Pablo F. Castro, Thomas S.E. Maibaum, and Nazareno Aguirre

Author Index 443

From Z to B and then Event-B: Assigning Proofs to Meaningful Programs

Jean-Raymond Abrial

Marseille, France
jrabrial@neuf.fr

The very first paper on Z [1] was published in 1980 (at the time, the name Z was not "invented"), then the book on the B method [2] was published in 1996, and, finally, the book on Event-B [3] was published in 2010. So, 30 years separate Z from Event-B. It is thus clear that I spent a significant time of my scientific professional life working with the same kind of subject in mind, roughly speaking specification languages. I do not know whether this kind of addiction is good or bad, but what I know is that I enjoyed it a lot.

So, I was very pleased when the organizers of iFM 2013, Luigia Petre and Einar Broch Johnsen, invited me to give a talk at the conference and at the same time suggested that I can give a presentation on "Z to Event-B".

Since I am the main contributor (at least initially) of these three topics (Z, B, and Event-B), there are certainly some common features that are interesting to put forward in the three of them. But at the same time, during these thirty years, it was also possible to gradually envisage some evolution from the first to the last.

The main purpose of this presentation is thus to clarify this. This paper does not contain a description of Z, B, or Event-B, as well as that of the corresponding tools that have been developed over the years. Readers interested by this can access the literature.

In the main part of the paper, I will rather present a historical account. The idea is to explain how all this has slowly emerged as an intellectual evolution. I will also try to make clear what are the external ideas and events (there are many) that influenced this evolution. Then I will try to make a synthesis and present what is common in these three topics and also what makes them different from each other. This will take place in two Appendices.

The Way It All Started: Green

In the seventies, I was a member to the "Green" team developing the programming language that later became ADA. During that time, I was probably one of the first persons to write programs in "Green": this is because I thought it was necessary to try some proposed programming features in some significant programs before incorporating them into a language. I must say that I was a bit frustrated at the time, and this was for two reasons.

My first frustration came from my colleagues of the "Green" team: they did not understand why I was so busy with this "programming" activity of mine. In fact, they were more interested in discussing some fine points of the language in a way that was, in my opinion, too far from the future job of working programmers.

My second, more important, frustration came from the programming examples that I decided to use for these experiments. How could I discover that the written programs were “correct”? More precisely, what does it means for a program to be “correct”? So, before engaging in the writing of these experimental programs, I decided to write little notes in English. Their purpose was precisely to clarify this point about “correctness”.

These notes supposed to describe what each future program was intended to *achieve*. In the case of a simple sequential programming experiment like a sorting program, this was rather straightforward. However, in the case of more elaborate experiments, I found it to be more difficult. In particular, this was the case on examples dealing with embedded systems which were the main intended target of the “Green” language.

About Assertions

Of course, I was aware of the effort made by Tony Hoare whose famous 1969 paper [8] was already well-known among researchers. But it seemed to me that writing assertions within a program and eventually proving that these assertions were indeed true was not enough, for the reason that the program and the assertions are written *simultaneously*.

My view was that one should have something at one’s disposal *before* writing a program, so that it would then be possible to compare fruitfully the program with something else. The most important word used in the previous statement is “compare”. This word raises some difficulties: how can I “compare” a program written in a formal programming language with a note written in English and how can I prove properties about such comparisons?

Looking for a Notation

Then emerged slowly the idea to write the mentioned “notes” describing the properties of the future program achievements, not in English, but in a more formal language so that the comparison between the program and the “notes” could be made more effective. Clearly the formal language in question could not be the programming language itself because this would then lead to some vicious circle. It should be noted however that in some programming languages, assertions are written using the programming language itself (with possible side effects...): an obvious misinterpretation. So, the question was: which notation shall we use to write such “notes”?

It happened that, at this time, I was very interested in reading the treatise [12] called “Théorie des Ensembles” (Éléments de Mathématiques) by Nicolas Bourbaki (this is the collective name of a group of French mathematicians). To be honest, I must admit that it took me a long time to become familiar with only the first 50 pages of this book. But I was an enthusiastic beginner and

I thought that the way first-order predicate calculus and set theory were used in this book was quite useful to introduce the main initial working concepts of mathematics (predicates, sets, binary relations, functions and their basic properties, etc.), although sometimes it was done in a complicated way in the book (to my subjective taste).

Proving

Putting together these investigations for a formal notation and my interest in this book was just a matter of time: it became clearer every day. What interested me in the Bourbaki book was not only the notation itself but also the way it was used to perform proofs (although such proofs are written in a rather informal way in the book). It slowly became clear to me that a mathematical notation is not only used to "write" mathematics but rather, and more importantly, as a way to write down proofs that can be trusted by the community of mathematicians (although this very difficult problem has a long history).

The usage of predicate calculus and that of a set-theoretic notation has thus a dual advantage for writing these famous notes. First, it will allow us to write them in a way that will be well understood by people with a normal scientific background. Second, it will allow us to do some formal proofs in a rigorous fashion. This second advantage helps us to give now more precision on how we can "compare" the notes and the programs: namely by doing some mathematical proofs.

All this was more or less perceived at the end of the seventies. But then it took more than thirty years to be fully understood (by me) and practically implemented on tools (by others and by me) ... and this is not even finished yet!

People Disliked It

When writing the first paper on Z and starting to spread the idea of using a set-theoretic notation to write a specification before writing a program, I received sometimes very negative answers. Some days at that time, I was invited to give a series of lectures at a Summer School but when the organizers saw the subject of my presentation, they canceled my visit, telling me that this subject was not relevant to computer scientists.

I discovered then that set theory had a bad reputation among computer scientists in Academia. I think the reason is that people are very afraid by the formal definition of it in, say, Zermelo-Fraenkel axiomatization (I agree). Quite often, computer science academic people confuses mathematics and computation, although in many working programs recursive and inductive definitions are totally absent.

On the one hand, computer scientists prefer to use predicate calculus only in specifications. But in doing that they still quite often confuse predicates (a logical concept) and boolean expressions (a programming concept). People think that mathematics has a semantics. Programming languages have of course a semantics, because programs can be executed. But mathematics has no semantics because it is not executed, it is only used to support modeling and proofs in various other disciplines (computer science is one of them).

On the other hand, I also noticed during this time, that professional mathematicians are not good at logic. To figure that out, it is very instructive to see how the "Intermediate Value Theorem" (a special case of it was proposed by Bolzano in1817) is "proved" in textbooks. The apparent difficulty comes from the fact that in this proof one needs to use the completeness axiom of real numbers and also the classical definition of the continuity of a real function. Both this theorem and this definition are formally defined by some heavy predicates involving both universal and existential quantifications. The proofs of the "Intermediate Value Theorem" that can be seen in the literature clearly show that their authors do not master predicate calculus: obscure wordings replace clear treatments. However, it is not at all difficult to have a perfect and readable proof provided it is made by someone who is "fluent" in first-order predicate calculus.

So, I was clearly in a bad situation: I wanted to reconcile computer scientists and mathematicians but, apparently, these two communities do not understand each other very well.

Oxford: Z

Fortunately not all people were against this approach: by the end of the seventies, I was invited by Tony Hoare to come to Oxford at the Programming Research Group, where I found a very open atmosphere and was able to develop further the notation with comprehensive colleagues: Bernard Sufrin, Tim Clement, and Ib Sorensen.

At the time, the emphasis was not put so much on proving. We were too busy trying to extract from the classical Zermelo-Fraenkel set theory the minimal notational constructs that could be used for writing program specifications.

However, we were very soon convinced that this notation should be an open one, because we could not clearly imagine what could be the necessary mathematical concepts that might be useful in writing specifications. The result of this initial work in Oxford was a very general open and powerful notation, Z, that is still presently in use in various places (mainly in Universities).

More Investigations: B

At the beginning of the eighties, I decided to come back to France and depart a bit from the Z community that was starting to grow. This community was quite active. People added very interesting features to Z: the schema and its calculus, and some techniques (using schemas) were developed to turn the notation into one for developing sequential programs.

I was interested by what was done there but also found that we were lacking an essential powerful proving system. I doubted that one could be developed out of such a general language as Z was. I also found that the relationships between different schemas was too weak. My new investigations went into four directions:

1. Could we simplify the mathematical notation of Z but nevertheless keep it powerful enough so that it could be used to develop significant programs?
2. Could we structure the notation into building blocks being able to mimic what happens in classical programming languages: modular languages and object oriented languages?
3. Could we incorporate into the notation some techniques borrowed from the refinement calculus of Ralph Back [6] and others (Tony Hoare [9], Cliff Jones [10], and Carroll Morgan [11]), so that the construction of large programs could be envisaged in a practical and gradual way?
4. Could we develop some powerful proving system that could be used in practice?

What became clear at the time was that the relationship between the mentioned "notes" and the corresponding program makes the former more and more important. Incorporating refinements into the pictures makes it possible now to investigate the following:

5. Could it be possible to remove completely the human programming and replace it by an automatic procedure performed on the last refinement of the specification?

With all this in mind, B was launched. But, of course, it took many more years for these ideas to be pushed enough until B could become a practical and industrial technique. This was clearly not at all the case at the time.

First Contact with Parisian Metro: RER

In the eighties, RATP (the Parisian Metro authority) was modernizing one of its RER lines (a special line from central Paris to and from some suburb). The idea was to have the driver following the instruction of a computer rather than looking at traffic lights. A huge embedded system was developed with computers on board the trains and on the wayside either. But when all this was completed, people in charge of the project were suddenly very afraid to sign the final authorization. This was because they were having some doubts about the correctness of the system. One could imagine the consequences of a sudden program crash. I was very surprised to be asked by RATP to perform a technical audit of this system. Among others, I had to answer the following question:

> Are we sure that the technique used to validate this system makes us confident that it will satisfy its specification?

For three weeks I met many people, essentially engineers in charge of developing this system and also those (different) in charge of testing it. I met extremely professional people who impressed me a lot. I had nothing against the verification technique they were using, essentially some heavy testing. But I had nevertheless a problem because I was not shown any clear specifications of this system. So, something was not clear to me: *against what* were these people testing?

When I presented my report to RATP and others, I said that I was very sorry but that I could not answer the question they asked me because I had not seen the specification. There was some kind of a shock in the audience. People were very nervous and said that I was not telling the truth. But the lack of specification documents (and design documents as well) was clearly established, so, at the end, they could not disagree.

As a result, and certainly for some other reasons, RATP was courageous enough and decided to postpone the starting of the new RER semi-automatic system for one year. Of course, they were not very happy to do so, because the Parisian Metro is considered to be one of the safest in the world: they had to recognize that this was not the case here.

More Contacts with Parisian Metro: Line 14

After the audit mentioned in the previous section, RATP asked me whether I could give them a course on how to write formal specifications. Needless to say, I was not at all at ease to give a positive answer, because B (this name even did not existed yet) was in its infancy and Z was not, in my view, the right approach to their problem. Nevertheless, I tried to do my best and I gave a course using a formal notation that was “in between” Z and B, clearly something that was not very satisfactory. Curiously enough, in spite of these approximations, the course was well received by RATP and by some industry people also attending the course, and, maybe more importantly, it helped me to clarify many things in B. During the course, we studied some part of the RER system (the structure and properties of a metro line) and tried to build a formal model of it.

After that, RATP was silent for a certain time: I thought that finally they were not interested by this approach. During that time, I tried to develop B further by stabilizing the retained notation and by starting to build a draft tool. The final notation was not an open one as Z was. A simple typing system was set up, and some structuring devices (probably too much) were developed as this was missing in Z. All this was done as I was a consultant for BP in the UK.

During this time I figured out how important it was to have some solid industrial contacts in order to develop some notation that could be used in practical developments. After more than a year, RATP contacted me again. They explained that they intended to develop a new metro line within Paris (Line 14, also called the Meteor Line) and that this line would be entirely automatic (without drivers). So far so good. But then they asked me whether B could be used to develop a part of the corresponding computerized system (the safety critical part). What was my feeling about it?

It was very difficult to give a positive answer: the definition of B was in principle stabilized but no experience was available yet about the usefulness of all this, and moreover, the tool was in its infancy (in particular the proving tool).

They suggested that the usage of B and that of the corresponding "tool" could be done in parallel with a more classical development. That seemed to me to be very reasonable, but probably quite costly! The company Matra Transport (now part of Siemens) was awarded the contract for developing this driverless train system. Of course, people in charge of this company were heavily against using B, but RATP convinced them (mainly financially) to go ahead with it (in parallel with a classical development as said before).

After some difficulties, a French software house, Steria, was in charge of developing further the tool for B (also funded by RATP). It was given the name "Atelier B". To my astonishment, this organization made of RATP, Matra Transport, and Steria worked well. At some point, RATP even decided to ask Matra Transport to cancel completely unit tests and also integration tests (both of them are very costly) because they felt confident enough in the proofs that were done with the tool. This saved a lot of money which was then used instead to perform more elaborate testing at a global system level.

In October 1998, Line 14 was launched. Since then, it has not suffered any software bugs. It has later been extended on both ends very easily. I must admit that I am always very impressed (and a bit anxious) to embark in a metro without driver running at full speed in the tunnels ... but the Parisians like it a lot: Line 14 is always very crowded.

Some Figures

In the Line 14 metro system, the software is not entirely developed with B, only those parts dealing with safety critical constraints are. On the whole, 86,000 lines of ADA code had been produced entirely automatically from the B development. Of course, it was totally forbidden to touch a single line of the code of this part. This B development required 27,800 proofs to be performed. Among them, 8.1% were interactive proofs. Such proofs are still done with the tool, but they require a human intervention. This corresponded to a work load of 7.1 man/months. The corresponding price is interesting as it can be compared with that of testing that had been canceled: a clear advantage to proving over testing.

What is interesting to note here is that the Matra Transport engineers had no difficulty to adapt to this technology that was entirely new for them. In particular, they were able to perform interactive proofs after a rather short updating. Moreover, the integration of the proof effort within their modeling work was very positive, difficulties in performing proofs became a sign that the modeling could be restructured: improving the modeling has a positive effect on the automatization of the proofs.

Ups and Downs

Since then, the usage of B has been expanded world wide in many other metro systems: in New York City, in South America, in Europe, in China, etc. Here are three projects developed with B in France:

1. Another Parisian metro line (Line 1) is now completely automatic and works with B.

2. The shuttle in Charles de Gaulle airport is automatic and is also using B. For that system, 158,000 lines of ADA were produced automatically. This required 43,600 proofs among which 3.3% were interactive, that is 4.6 man/months (the difference with the figures for Line 14 project, as mentioned in the previous section, is due to some improvment in the tool).

3. A line of the Lyon metro has been renovated and uses B.

The Atelier B tool has been developed further by Clearsy, specially its prover. Other tools have been developed by Siemens to partially automatize the refinement process. Finally, two French software houses are making successful business with B: Clearsy and Systerel. All this is the positive side.

But there is also a negative side that is quite important. Curiously enough, to the best of my knowledge, the usage of B is totally absent in other advanced industrial domains: automotive, aircraft, space, nuclear industries, etc. To my opinion, B could have been used equally well in these domains. In fact, people there strongly object to use it. They claimed to have many reasons to do so: in particular, they said that their engineers could not perform interactive proofs. They also claimed that B is too far from their engineers technical culture. They took other options so that it is certainly the case that there will probably be no progress in using a formal method such as B for the next 15 years in these areas.

On the negative part, it is also interesting to note that B was very much disregarded initially by Siemens when this company purchased Matra Transport (it became Siemens Transport). Also in RATP itself, some lobbies were strongly against B. One of their arguments was that because of the automatization of proofs and programming then the engineers would be less involved in their job. I heard also the argument that because B was a success in the Line 14 project then one has to think of a different approach! In both cases, after some years, B was still being used fruitfully...

System Modeling

At the turn of the century, after the success of the Line 14 project of Paris metro, I was worried about one question, which is the following. In this project and in many similar ones, some initial studies are performed by, so-called, system engineers. They determine the main structure of the system, mainly its components and their relationship. They eventually deliver an analysis out of which one defines the informal specifications of the various components of the future system. The formal development undertaken with B (or with any other similar approach) starts at this point only. This is then done on each of these components independently.

Now, the question is: how about these system studies? I am not claiming that the system engineers are doing a bad job, they are usually extremely good and professional, but, clearly, if mistakes are made during this preliminary phase, then the formal developments undertaken afterwards will probably not discover them.

In the case of a train system, such studies are responsible for ensuring that no two trains can hit each other when they both circulate on a complex rail network containing many dynamic points (switches in US English) allowing train to change rails. In other words, they guarantee that many trains can circulate simultaneously on the rail network in a safe way. As can be seen, this is not a simple task: the computer of each train has to communicate with the wayside computers receiving information about various trains circulating in their neighborhood and information about the points' position in the rails under the supervision of the wayside computers.

So the idea is to develop a formal approach for such system studies as well. We would like these studies to be concluded by formal proofs, thus being possibly safer than those done "manually". Clearly the formalism to be used here had to be different from that used for developing software as is done with B. We have to model an entire system where some components will eventually turn into software controllers whereas others are models of the physical components pertaining to the, so-called, environment of the various software parts.

In other words, we have to model a situation where the "computation" is essentially distributed. We also have to take into account the cases where one of the components fails. Finally, we have to consider complex timing constraints. All this make it necessary to start a new formalism that inherited a lot from B but was nevertheless different from it: Event-B [3].

A European Effort: Event-B and the Rodin Platform

At that time, I was made aware by Michael Butler of the work done in the late eighties by Ralph Back and Reino Kurki-Suonio on Action System [5]. I became fascinated by the excellent ideas contained in their approach and figured out that many of them could be borrowed and incorporated into B for making such formal system modeling possible. It happened that B, as it what at the time, was rich enough to be used for experimenting these ideas. This was done for many years.

From 2002 until now, this new developmentt was funded by the European Commission under four different successive European projects: Matisse, Rodin, Deploy, and Advance. This effort of the European Commission is quite remakable. Some European Universities were involved: the University of Southampton and Newcastle University in the UK, Aabo Akademi in Finland, ETH-Zuerich in Switzerland, and the University of Duesseldorf in Germany. Likewise, some industrial companies were involved among which are Siemens, Bosch, SAP, Alstom, Clearsy, and Systerel.

I also received some very important help and scientific support from Dominique Cansell and Dominique Méry.

These European projects evolved in three directions: first the development of Event-B itself, second the construction of a tool, the Rodin Platform [15], and third, some industrial case studies (such case studies were not completely satisfactory).

The Rodin tool is built on top of Eclipse. This allowed us to have a platform that is extensible by means of, so-called, plug-ins. The development of the Rodin Platform started at ETH Zuerich during the Rodin project. The technical leader of the project was Laurent Voisin working with by Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and François Terrier. It was extended during the Deploy project at Systerel, still under the guidance of Laurent Voisin working this time with Nicolas Beauger, Thomas Muller, and Christophe Métayer.

Many plug-ins for the Rodin Platform were developed at the University of Southampton under the guidance of Michael Butler (among which are UML-B, EventB-ADA,Theory), and at the University of Duesseldorf under the guidance of Michael Leuschel (ProB, ProR).

The net result of all these efforts is that we now have a very rich software tool [15] that can be made available for free to anyone. Courses on Event-B and practical works with the Rodin Platform are taught in many parts of the world: Australia, Japan, China, Malaysia, India, many places in Europe, South and North America, North Africa.

Putting B and Event-B Together

The next step is now to try both Event-B and B in the same industrial project. The former being used to perform the system studies followed by the latter used to implement the software controller. This has not fully happen yet although I am quite confident that it could be done.

There has been an attempt, but the industrial partner was not ready to fully cope with this dual approach, although he was quite convinced and happy with the usage of B, which was very successful.

The Last Evolution: Hybrid Systems

In recent years, several groups, mainly in Shanghai (East China Normal University) and in the UK (Manchester University) [14] are interested in studying how Event-B and the Rodin platform could be able to cope with hybrid systems.

So far, B and Event-B were used to model discrete systems only. But hybrid systems are becoming more and more important, especially in the development of embedded systems where the problem is to control, with some discrete interventions, an external situation that is supposed to evolve continuously.

Here again, among a very important literature, the work of Ralph Back and his colleagues Luigia Petre and Ivan Porres [7] is quite important. They extended Action System to become Continuous Action system. This is done by generalizing timeless variables to be time function variables recording their future evolution as time goes. This generalization can be incorporated into models done with Event-B without too much difficulties [4].

However, when such functions are not defined directly but rather indirectly by means of unsolvable differential equations, things are becoming more complicated. One has thus to prove some properties of the solution of a differential equation without solving it. This topic is still presently a research subject.

Some Concluding Remarks

In this paper, I covered thirty years of history concerning Z, B, and Event-B. The sociological adventure of all this is quite interesting. The first remark that could be made is that more and more people and institutions were involved over the years. On can also notice that people involvement was not so much "linear" on the short term (it could stop temporarily) but it does increase regularly in the long term.

The situation with Event-B and the Rodin Platform is quite different from that of B and the Atelier B tool. Event-B and the Rodin Platform are largely used in Academia and almost not used in Industry (with the exception of Hitachi in Japan). On the contrary, B and the Atelier B tool are used in many industrial projects but not so much in Academia. It could be interesting to investigate why.

I think that the main reason is due to the different kinds of funding that were used. In the case of B and Atelier B, the main funding came from RATP together with the development of a huge industrial project: Line 14.

Some lessons have to be learned from this. It seems to me that a practical research could be partially funded by taking a very small percentage of the money spent in large industrial projects. This is what has happened for the funding to B and Atelier B. The same thing happens with some piece of art in France and certainly in other countries as well. For instance, when the French government decides to fund the construction of, say, a hospital, then a small amount of the funding is devoted to pay an artist to erect a sculpture in front of the hospital. I think that there is even a law enforcing this.

Why not to have the same kind of law for the funding of a practical research? There will be many positive outcomes from doing this: the research project will last as long as the industrial project does. In my opinion, it is important, although sometimes difficult, to stop a research project, here it would be implicit from the beginning. The outcome of the research project (if successful) can be incorporated into the industrial project. People involved in the research project are very committed because they are willing to see their results being used in the industrial project and later elsewhere.

In the case of Event-B and the Rodin Platform, the funding came from the European Commission. In these cases we observed, as mentioned earlier, that the industrial case studies were not very successful. The reason, I think, for these relative failures came from the fact that the involved parts of the industries were R&D units, not business units. It is well known fact that in large corporations, the relationship between these entities is rather delicate: usually, works done in R&D departments have some difficulties to be eventually incorporated into business units.

On the other hands, the academic results of these European projects is extremely successful (Event-B, the Rodin Platform). So, the European funding was here very positive. As said earlier, many courses on this material are now distributed in many places over the world and many PhD thesis were awarded. The influence on industry can then take more time when these educated people will eventually reach industry. This is a long-term human investment very much in the spirit of the European Commission.

Acknowledgments. I would like to thank very much some people who read initial drafts of this paper providing encouragements and useful comments: Richard Banach, Michael Butler, Thai Son Hoang, Luigia Petre, Ken Robinson, and Bernard Sufrin.

References

1. Abrial, J.-R., Schuman, S.A., Meyer, B.: Specification Language. On the Construction of Programs (1980)
2. Abrial, J.-R.: The B-book: assigning programs to meanings. Cambridge University Press (1996)
3. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
4. Abrial, J.-R., Su, W., Zhu, H.: Formalizing Hybrid Systems with Event-B. ABZ (2012)
5. Back, R.J., Kurki-Suonio, R.: Distributed Cooperation with Action Systems. ACM Transaction on Programming Languages and Systems (1988)
6. Back, R.J.: A Calculus of Refinements for Program Derivations. Acta Informatica (1988)
7. Back, R.-J., Petre, L., Porres, I.: Generalizing action systems to hybrid systems. In: Joseph, M. (ed.) FTRTFT 2000. LNCS, vol. 1926, p. 202. Springer, Heidelberg (2000)
8. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. CACM (1969)
9. Hoare, C.A.R.: Proof of Correctness of Data Representations. Acta Informatica (1972)
10. Jones, C.B.: Software Development: A Rigorous Approach. Prentice Hall International (1980)
11. Morgan, C.C.: Programming from specifications. Prentice Hall International (1990)
12. Bourbaki, N.: Théorie des Ensembles. Hermann (1970)
13. Maamria, I.: Towards a Practically Extensible Event-B Methodology. PhD Thesis. The University of Southampton (2012)
14. Banach, R., Zhu, H., Su, W., Wu, X.: Continuous Behaviour in Event-B: A Sketch. ABZ (2012)
15. `http://www.event-b.org`

Appendix A: What Is Common in Z, B, and Event-B: the Set Notation

The mathematical notation used in Z, B and Event-B is that of set theory. It is often said that Z is named after Zermelo-Fraenkel set theory. People could then think that the Z (and B) notation was directly dictated by this set theory. In my opinion, this is far from being true. In order to illustrate this, let me present here a mail conversation I had some times ago with Prof. Freek Wiedijk. Here is part of a mail I received from him:

> I was talking to Josef Urban, and he claimed that someone told him last week (at Dagstuhl) that B has the full strength of ZFC set theory. Now I had the impression that although B is very much in the *spirit* of ZFC set theory, it is logically slightly weaker. I don't remember where that impression came from, though. (Maybe it was related to the type system?)
> Can you tell us who's right about this? I would very much like to know!

Here was my answer:

> Let me explain what we have in Event-B[1] (see chapter 9 of my book).
>
> **Set Theory**
> The set theory we use in Event-B is a TYPED (see below) set theory made of the following axioms:
> 1. Extensionality: $A = B \Leftrightarrow \forall x \cdot x \in A \Leftrightarrow x \in B$
> 2. Power set: $T \in \mathbb{P}(S) \Leftrightarrow \forall x \cdot x \in T \Rightarrow x \in S$
> 3. Comprehension: $x \in \{y \,|\, y \in S \wedge P(y)\} \Leftrightarrow x \in S \wedge P(x)$
> 4. Cartesian product: $x \mapsto y \in S \times T \Leftrightarrow x \in S \wedge y \in T$ (see below for the notation of the pair of x and y)
> 5. The axiom of choice can be added if necessary by defining in the language a (polymorphic) choice function for each non-empty set.
>
> **Remarks**
> r1: It does not contain the foundation axiom. It is implicit from the syntax.
> r2: It does not contain the union axiom. It is forbidden by the typing.
> r3: It does not contain the infinity axiom as the natural numbers and the integers are axiomatized directly with the Peano axioms and the extension to negative numbers.
> r4: The Cartesian product axiom is necessary as the usual Kuratowski construction of the pair ($(x, y) = \{\{x\}, \{x, y\}\}$) is forbidden by the typing.
> r5: In fact, the pair is axiomatized OUTSIDE SET THEORY as it it necessary in predicate calculus with equality: $x \mapsto y = z \mapsto t \Leftrightarrow x = z \wedge y = t$ (we denote the pair of x and y as $x \mapsto y$).

[1] What follows is equally valid for Z and B.

About Typing
When defining a problem, we start with statements like this "Let S, T, ... be given sets ..." Within such a statement, the BASIC TYPES are the sets S, T, ... together with $\mathbb{Z}$, the set of Integers. Type constructors are:
1. Cartesian product of types
2. Power of types.
Type checking (a procedural treatment) must ensure that all expressions in a formal statement are WELL-TYPED. It means that each expression belongs to a basic type or belong to a set defined (recursively) from basic types and the type constructors. If it is not the case, then the statement is rejected for future treatment.

Well-Definedness
Besides type checking, another initial treatment is performed on a formal statement, namely checking that expressions and predicates are WELL-DEFINED. This treatment may require some proofs (it is NOT procedural as type checking is).
The most common well-definedness checking is that in an expression such as $f(x)$ (where f is a partial function) then x must belong to the domain of f. Others are no division by 0, min and max are well-defined on some numerical sets ...
In a proof, after well-definedness checking, one can assume that expressions remain well defined unless one introduces new expressions as in a cut treatment (in this case, well-definedness has to be performed in the middle of a proof).

Answering Your Question
I agree with your wording: "B is very much in the *spirit* of ZFC set theory". Now the question is: can we do the same as in full ZFC? We have tried many set theoretic theorems such as (see the end of the last chapter of my book): Tarski's fix point theorem, Cantor-Bernstein theorem, Zermelo's theorem, etc. We were able to prove them all within our set theory. However ordinals cannot be defined explicitly because of typing.

Finally, here is the reply sent by Prof. Freek Wiedijk

Many thanks for clarifying to us how the set theory of Event-B relates to ZFC!
What I find suprising is that you claim that the fact that it is a bit weaker (no union axiom, no replacement, no way to define the ordinals) is caused by the fact that everything is typed. However, in the Mizar system one *does* have full ZFC (even a bit more), and there everything is typed too.
But I guess that maybe you don't like to have a more complicated type system (like the one in Mizar, with its rather involved subtyping rules) for good reasons?

Here is my final mail in reply to this one:

> Yes, the typing we use is very simple (basic types + (recursively) cartesian product of types and power of types). This was decided on purpose so that the typing procedure is also very simple (does not require any proof) ... and, as we find it, sufficient for our purpose.

Appendix B: What Is Different in Z, B, and Event-B: Their Purpose

The initial purpose of Z was to identify a set-theoretic notation to be used as a medium for writing formal specifications of programs. The result is a very powerful generic notation.

The initial purpose of B was to develop a notation able to be used for the specification and development of large software. The set notation used, although in the same spirit as the one used in Z, is in fact less general and not generic. It was felt however to be sufficient for the intended purpose. The main structure is the so-called "machine" containing variables, invariant and operations defined by pre- and post-conditions. A machine can be refined. It can also import another machine for an implementation. The notation is thus close (although more general) to that of a modular programming language. Refinement is defined by weakening the pre-condition and strengthening the post-condition of operations.

The Atelier B tool (proprietary of Clearsy but freely distributed to Universities) allows one to define machines, refines them and prove the corresponding proof obligations (invariant preservation and correct refinement).

The initial purpose of Event-B is to have a notation able to be used for modeling distributed systems. As for B, the main structure is also a "machine". However a machine does not define a programming module as in B but rather contains a set of events defined by a guard and an action (assignment, possibly non-deterministic of variables). Refinement is defined by strengthening the guard and the actions.

The Rodin Platform [15] is a free software, and, like Atelier B for B, allows one to define machines, refines them and prove the corresponding proof obligations (invariant preservation and correct refinement). This platform sits on top of Eclipse. It can be extended by plug-ins.

The initial set-theoretic notation of Event-B is a little less general that the one used in B. However, the usage of the "Theory" plug-in [13] under the Rodin Platform allows one to freely extend the mathematical language in a polymorphic way.

Systems Design Guided by Progress Concerns

Simon Hudon[1] and Thai Son Hoang[2]

[1] Department of Computer Science, York University, Toronto, Canada
simon@cse.yorku.ca
[2] Institute of Information Security, ETH-Zurich, Switzerland
htson@inf.ethz.ch

Abstract. We present Unit-B, a formal method inspired by Event-B and UNITY, for designing systems via step-wise refinement preserving both safety and liveness properties. In particular, we introduce the notion of coarse- and fine-schedules for events, a generalisation of weak- and strong-fairness assumptions. We propose proof rules for reasoning about progress properties related to the schedules. Furthermore, we develop techniques for refining systems by adapting event schedules such that liveness properties are preserved. We illustrate our approach by an example to show that Unit-B developments can be guided by both safety and liveness requirements.

Keywords: progress properties, refinement, fairness, scheduling, Unit-B.

1 Introduction

Developing systems satisfying their desirable properties is a non-trivial task. Formal methods have been seen as a solution to the problem. Given the increasing complexity of systems, many formal methods adopt refinement techniques, where systems are developed step-by-step in a property-preserving manner. In this way, a system's details are gradually introduced into its design in a hierarchical development.

System properties are often put into two classes: *safety* and *liveness* [10]. A safety property ensures that undesirable behaviours will never happen during the system executions. A liveness property guarantees that eventually desirable behaviours will happen. Ideally, systems should be developed in such a way that they satisfy both their safety and liveness properties. Although safety properties are often considered the most important ones, we argue that having *live* systems is also important. A system that is safe but not live is useless. For example, consider an elevator system that does not move. Such an elevator system is safe (nobody could ever get hurt), yet useless. According to a survey [6], liveness properties (in terms of *existence* and *progress*) amount to 45% of the overall system properties.

In most refinement-based development methods such as (B, Event-B, VDM, Z) the focus is on preserving safety properties. A possible problem for such safety-oriented methods is that when applying them to design a system, we can make the design so safe that it becomes unusable. It is hence our aim to design a refinement framework preserving both safety and liveness properties.

Some modelling methods such as UNITY [3], include the capability of reasoning about liveness properties. In UNITY, there is a clear distinction between specifications (temporal properties) and programs (transition systems). Refinement in UNITY involves transforming specifications according to the UNITY logic. At the end of the refinement process, one obtains several temporal properties which then can be implemented by some program fragments according to well-defined rules. As a result, programs (transition systems) in UNITY are not part of the design, they are the output of the refinement process. A disadvantage of this approach is that the transformation of temporal properties can make the choice of refinements hard to understand. In order to overcome this limitation, we unified the notion of specification and that of program, making smoother the transition from one to the other.

In this paper, we present a formal method, namely Unit-B [8], inspired by UNITY [3] and Event-B [1]. We borrow the ideas of system development from Event-B, in which a series of models is constructed and linked by refinement relationships. The temporal logic that we use to specify and to reason about progress properties is based on UNITY.

The main attraction of our method is that it incorporates the reasoning about safety and liveness properties within a single refinement framework. Furthermore, our approach features the novel notions of coarse- and fine-schedules, a generalisation of the standard weak- and strong-fairness assumptions. They allow us (1) to reason about the satisfiability of progress properties by a given model, and (2) to refine a given model while preserving liveness properties. This makes it possible in Unit-B to introduce liveness properties at any stage of the development process. Subsequently, not only does it rule out any design that would be too conservative, but it also justifies design decisions. As a result, liveness properties, in particular progress properties, act as a design guideline for developing systems.

We give a semantics for Unit-B models and their properties using computation calculus [5]. This enables us to formally prove the rules for reasoning about properties and refinement relationship in Unit-B.

Structure. The rest of the paper is organised as follows. In Section 2, we review Dijkstra's computation calculus [5] which we used to formulate our semantics and design our proofs. We follow with a description of the Unit-B method (Section 3). The method and its refinement rules are demonstrated by an example in Section 4. We summarise our work in Section 5 including discussion about related work and future work.

2 Background: Computation Calculus

This section gives a brief introduction to computation calculus, based on [5]. Let $\mathcal{S}$ be the state space: a non-empty set of "states". Let $\mathcal{C}$ be the computation space: a set of non-empty (finite or infinite) sequences of states ("computations"). The set of computation predicates $CPred$ is defined as follows.

Definition 1 (Computation Predicate). $CPred = \mathcal{C} \to \mathbb{B}$, *i.e. functions from computations to Booleans.*

The standard boolean operators of the predicate calculus are lifted, i.e. extended to apply to $CPred$. For example, assuming $s, t \in CPred$ and $\tau \in \mathcal{C}$, we have,[1]

$$(s \Rightarrow t).\tau \;\equiv\; (s.\tau \Rightarrow t.\tau) \quad (1) \qquad \langle \forall\, i :: \; s.i \rangle.\tau \;\equiv\; \langle \forall\, i :: \; s.i.\tau \rangle \; . \quad (2)$$

The everywhere-operator quantifies universally over all computations, i.e.

$$[\, s \,] \;\equiv\; \langle \forall\, \tau :: \; s.\tau \rangle \tag{3}$$

Whenever there are no risks of ambiguity, we shall use $s = t$ as a shorthand for $[\, s \equiv t \,]$ for computation predicates s, t.

Postulate 1. *$CPred$ is a predicate algebra.*

A consequence of Postulate 1 is that $CPred$ satisfies all postulates for the predicate calculus as defined in [4]. In particular, $\mathbf{true}$ (maps all computations to TRUE) and $\mathbf{false}$ (maps all computations to FALSE) are the "top" and the "bottom" elements of the complete boolean lattice with the order $[\, \_ \Rightarrow \_ \,]$ specifying by these postulates. The lattice operations are denoted by various boolean operators including $\wedge, \vee, \neg, \Rightarrow$, etc.

The predicate algebra is extended with sequential composition as follows.

Definition 2 (Sequential Composition)

$$(s;t).\tau \;\equiv\; (\#\tau = \infty \wedge s.\tau) \;\vee\; \langle \exists\, n \colon \; n < \#\tau \colon \; s.(\tau \uparrow n{+}1) \wedge t.(\tau \downarrow n) \rangle \tag{4}$$

where $\#$, $\uparrow$ and $\downarrow$ denote sequence operations 'length'*,* 'take' *and* 'drop'*, respectively.*

Intuitively, a computation τ satisfies $s\,;t$ if either it is an infinite computation satisfying s, or there is a finite prefix of τ (i.e. $\tau \uparrow n{+}1$) satisfying s and the corresponding suffix $\tau \downarrow n$ (which overlaps with the prefix on one state) satisfying t.

In the course of reasoning using computation calculus, we make use of the distinction between infinite ("eternal") and finite computations. Two constants $\mathbf{E}, \mathbf{F} \in CPred$ have been defined for this purpose.

Definition 3 (Eternal and Finite Computations). *For any predicate s,*

$$\mathbf{E} = \mathbf{true}; \mathbf{false} \quad (5) \qquad s \textit{ is eternal} \;\equiv\; [\, s \Rightarrow \mathbf{E} \,] \quad (7)$$

$$\mathbf{F} = \neg\mathbf{E} \quad (6) \qquad s \textit{ is finite} \;\equiv\; [\, s \Rightarrow \mathbf{F} \,] \quad (8)$$

Given $\mathbf{F}$ the temporal "eventually" operator (i.e., $\Diamond$) can be formulated as $\mathbf{F}; s$. The "always" operator $\mathbf{G}$ is defined as the dual of the "eventually" operator.

Definition 4 (Always Operator). $\mathbf{G}\, s = \neg(\mathbf{F}; \neg s)$*, for any predicate s*

Important properties of $\mathbf{G}$ are that it is strengthening and monotonic. For any predicates s and t, we have:

$$[\, \mathbf{G}\, s \Rightarrow s \,] \, , \tag{9}$$

$$[\, s \Rightarrow t \,] \;\Rightarrow\; [\, \mathbf{G}\, s \Rightarrow \mathbf{G}\, t \,] \, , \tag{10}$$

$$[\, \mathbf{G}\,(s \Rightarrow t) \;\Rightarrow\; (\mathbf{G}\, s \Rightarrow \mathbf{G}\, t) \,] \, . \tag{11}$$

[1] In this paper, we use $f.x$ to denote the result of applying a function f to argument x. Function application is left-associative, so $f.x.y$ is the same as $(f.x).y$.

A constant $\mathbb{1}$ is defined as the (left- and right-) neutral element for sequential composition.

Definition 5 (Constant $\mathbb{1}$). *For any computation τ, $\mathbb{1}.\tau \;\equiv\; \#\tau = 1$*

State Predicates In fact, $\mathbb{1}$ is the characteristic predicate of the state space. Moreover, we choose not to distinguish between a single state and the singleton computation consisting of that state, which allows us to identify predicates of one state with the predicates that hold only for singleton computations. Let us denote the set of state predicates by $SPred$.

Definition 6 (State Predicate). *For any predicate p, $p \in SPred \;\equiv\; [\, p \Rightarrow \mathbb{1} \,]$.*

A consequence of this definition is that $SPred$ is also a complete boolean lattice with the order $[\, \_ \Rightarrow \_ \,]$, with $\mathbb{1}$ and **false** being the "top" and "bottom" elements. It inherits all the lattice operators that it is closed under: conjunction, disjunction, and existential quantification. The other lattice operations, i.e. negation and universal quantification, are defined by restricting the corresponding operators on $CPred$ to state predicates. We only use state predicate negation in this paper.

Definition 7 (State predicate negation $\sim$). *For any state predicate p, $\sim p = \neg p \wedge \mathbb{1}$.*

For a state predicate p, the set of computations with the initial state satisfying p is captured by $p\,;\mathbf{true}$: the weakest such predicate. A special notation $\bullet \;:\; SPred \to CPred$ is introduced to denote this predicate.

Definition 8 (Initially Operator). *For any state predicate p, $\bullet p = p\,;\mathbf{true}$*

This entails the validity of the following rule, which we will use anonymously in the rest of the paper: for p, q two *state predicates*, $p\,;q \;=\; p \wedge q$.

An important operator in LTL is the "next-time operator". This is captured in computation calculus by the notion of atomic computations: computations of length 2. A constant $\mathbf{X} \in CPred$ is defined for this purpose.

Definition 9 (Atomic Actions). *For any computation τ and predicate a,*

$$\mathbf{X}.\tau \;\equiv\; \#\tau = 2 \qquad (12)$$

$$a \textit{ is an atomic action} \;\equiv\; [\, a \Rightarrow \mathbf{X} \,] \qquad (13)$$

Given the above definition, the "next" operator can be expressed as $\mathbf{X}\,;s$ for arbitrary computation s.

3 The Unit-B Method

This section presents our contribution: the Unit-B method which is inspired by Event-B and UNITY. Similar to Event-B, Unit-B is aimed at the design of software systems by stepwise refinement. It differs from Event-B by the capability of reasoning about progress properties and its refinement-order which preserves liveness properties. It also differs from UNITY by unifying the notions of programs and specifications, allowing refinement of programs.

3.1 Syntax

Similar to Event-B, a Unit-B system is modelled by a transition system, where the state space is captured by variables v and the transitions are modelled by guarded events. Furthermore, Unit-B has additional assumptions on how the events should be scheduled. Using an Event-B-similar syntax, a Unit-B event has the following form:

$$\mathsf{e} \mathrel{\widehat{=}} \textbf{any}\ t\ \textbf{where}\ g.t.v\ \textbf{during}\ c.t.v\ \ \textbf{upon}\ f.t.v\ \ \textbf{then}\ s.t.v.v'\ \ \textbf{end}\ \ , \tag{14}$$

where t are the parameters, g is the *guard*, c is the *coarse-schedule*, f is the *fine-schedule*, and s is the *action* changing state variables v. The action is usually made up of several *assignments*, either deterministic ($:=$) or non-deterministic ($:|$). An event e with parameters t stands for multiple events. Each corresponds to several non-parameterised events $\mathsf{e}.t$, one for each possible value of the parameter t. Here g, c, f are state predicates. An event is said to be enabled when the guard g holds. The scheduling assumption of the event is represented by c and f as follows: if c holds for infinitely long and f holds infinitely often then the event is carried out infinitely often. An event without any scheduling assumption will have its coarse-schedule c equal to **false**. An event having only the coarse-schedule c will have the fine-schedule to be $\mathbb{1}$. Vice versa, an event having only the fine-schedule f will have the coarse-schedule to be $\mathbb{1}$.

In addition to the variables and the events, a model has an initialisation state predicate init constraining the initial value of the state variables. All computations of a model start from a state satisfying the initialisation and are such that, at every step, either one of its enabled events occurs or the state is unchanged, and each computation satisfies the scheduling assumptions of all events.

Properties of Unit-B models are captured by two types of properties: *safety* and *progress* (liveness).

3.2 Semantics

We are going to use computation calculus to give the semantics of Unit-B models. Let $\mathbf{M}$ be a Unit-B model containing a set of events of the form (14) and an initialisation predicate init. Since the action of the event can be described by a before-after predicate $s.t.v.v'$, it corresponds to an atomic action $\mathbf{S}.t \ = \ \langle \forall\, e ::\ \bullet(e = v)\ \Rightarrow\ \mathbf{X}\,; s.t.e.v\rangle$. Given that an event $\mathsf{e}.t$ can only be carried out when it is enabled, the effect of each event execution can therefore be formulated as follows: $act.(\mathsf{e}.t) = g.t\,;\ \mathbf{S}.t$. A special constant SKIP is used to denote the atomic action that does not change the state.

Definition 10 (Constant SKIP). SKIP.$\tau \ \equiv\ \#\tau = 2\ \wedge\ \tau.0 = \tau.1$, *for all traces* τ *(*$\tau.0$, $\tau.1$ *denotes the first two elements of* τ*).*

The semantics of $\mathbf{M}$ is given by a computation predicate $ex.\mathbf{M}$ which is a conjunction of a "safety part" $\mathit{saf}.\mathbf{M}$ and a "liveness part" $\mathit{live}.\mathbf{M}$, i.e.,

$$[\ ex.\mathbf{M}\ \equiv\ \mathit{saf}.\mathbf{M} \wedge \mathit{live}.\mathbf{M}\ ]\ . \tag{15}$$

A property represented by a formula s is satisfied by $\mathbf{M}$, if

$$[\ ex.\mathbf{M} \Rightarrow s\ ]\ . \tag{16}$$

Safety Below, we define the general form of one step of execution of model $\mathbf{M}$ and the *safety* constraints on its complete computations.

$$[\, step.\mathbf{M} \;\equiv\; \langle \exists\, \mathsf{e}, t \colon\ \mathsf{e}.t \in \mathbf{M} \colon\ act.(\mathsf{e}.t) \rangle \;\vee\; \text{SKIP} \,] \tag{17}$$

$$[\, saf.\mathbf{M} \;\equiv\; \bullet\, \mathsf{init} \wedge \mathbf{G}\,(step.\mathbf{M}\,;\, \mathbf{true}) \,] \tag{18}$$

Safety properties of the model are captured by *invariance* properties (also called *invariants*) and by *unless* properties.

An invariant $I.v$ is a state-properties that hold at every reachable state of the model. In order to prove that $I.v$ is an invariant of $\mathbf{M}$, we prove that $[\ ex.\mathbf{M} \;\Rightarrow\; \mathbf{G}\ \bullet I\ ]$. In particular, we rely solely on the safety part of the model to prove invariance properties, i.e., we prove $[\ saf.\mathbf{M} \;\Rightarrow\; \mathbf{G}\ \bullet I\ ]$. This leads to the well-known invariance principle.

$$\begin{array}{l} \quad [\, \mathsf{init} \Rightarrow I \,] \;\wedge\; [\, \langle \forall\, \mathsf{e}, t\colon\ \mathsf{e}.t \in \mathbf{M}\colon\ I\,; act.(\mathsf{e}.t) \;\Rightarrow\; \mathbf{X}; I \rangle \,] \\ \Rightarrow \\ \quad [\, saf.\mathbf{M} \Rightarrow \mathbf{G}\ \bullet I \,] \end{array} \tag{INV}$$

Invariance properties are important for reasoning about the correctness of the models since they limit the set of reachable states. In particular, invariance properties can be used as additional assumptions in proofs for progress properties.

The other important class of safety properties is defined by the *unless* operator **un**.

Definition 11 (un operator). *For all state predicates p and q,*

$$[\, (p\, \mathbf{\textit{un}}\, q) \;\;\equiv\;\; \mathbf{G}\,(\bullet p \;\Rightarrow\; (\mathbf{G}\ \bullet p)\,;(\mathbb{1} \vee \mathbf{X})\,;\bullet q) \,] \tag{19}$$

Informally, p **un** q is a safety property stating that if condition p holds then it will hold continuously unless q becomes true. The formula $(\mathbb{1} \vee \mathbf{X})$ is used in (19) to allow the last state where p holds and the state where q first holds to either be the same state or to immediately follow one another. The following theorem is used for proving that a Unit-B model satisfies an unless property.

Theorem 1 (Proving an un-property). *Consider a machine $\mathbf{M}$ and property p **un** q. If*

$$\langle \forall\, \mathsf{e}, t\colon\ \mathsf{e}.t \in \mathbf{M}\colon\ \mathbf{G}(\,(p \wedge {\sim} q); act.(\mathsf{e}.t); \mathbf{true} \;\;\Rightarrow\;\; \mathbf{X}; (p \vee q); \mathbf{true}\,)\rangle \tag{20}$$

then $[\ ex.\mathbf{M} \;\Rightarrow\; p\, \mathbf{\textit{un}}\, q\ ]$

Proof (Sketch). Condition (20) ensures that every event of $\mathbf{M}$ either maintains p or establishes q. By induction, we can see that the only way for p to become false after a state where it was true is that either q becomes true or that it was already true.

Liveness. For each event of the form (14), its schedule $sched.(\mathsf{e}.t)$ is formulated as follows, where c and f are the event's coarse- and fine-schedule, respectively.

$$[\, sched.(\mathsf{e}.t) \;\;\equiv\;\; \mathbf{G}\,(\mathbf{G}\ \bullet c \;\wedge\; \mathbf{G}\,\mathbf{F}\,; \bullet f \;\Rightarrow\; \mathbf{F}\,; f\,; act.(\mathsf{e}.t)\,; \mathbf{true}) \,]\,. \tag{21}$$

To ensure that the event $\mathsf{e}.t$ only occurs when it is enabled, we require the following *feasibility* condition:

$$[\, ex.\mathbf{M} \;\;\Rightarrow\;\; \mathbf{G}\ \bullet(c \wedge f \;\Rightarrow\; g) \,] \tag{SCH-FIS}$$

Our scheduling is a generalisation of the standard weak-fairness and strong-fairness assumptions. The standard *weak-fairness* assumption for event e (stating that if the event is enabled infinitely long then eventually it will be taken) can be formulated by using $c = g$ and $f = \mathbb{1}$. Similarly, the standard *strong-fairness* assumption for e (stating that if the event is enabled infinitely often then eventually it will be taken) can be formulated by using $c = \mathbb{1}$ and $f = g$.

$$[\ \mathrm{wf}.(\mathsf{e}.t) \equiv \mathbf{G}\,(\mathbf{G}\ \bullet g \ \Rightarrow\ \mathbf{F}; act.(\mathsf{e}.t); \mathbf{true})\ ] \tag{22}$$

$$[\ \mathrm{sf}.(\mathsf{e}.t) \equiv \mathbf{G}\,(\mathbf{G}\,\mathbf{F}; \bullet g \ \Rightarrow\ \mathbf{F}; act.(\mathsf{e}.t); \mathbf{true})\ ] \tag{23}$$

The liveness part of the model is the conjunction of the schedules for its events, i.e.,

$$[\ live.\mathbf{M} \ \equiv\ \langle \forall\, \mathsf{e}, t\colon\ \mathsf{e}.t \in \mathbf{M}\colon\ sched.(\mathsf{e}.t)\rangle\ ] \tag{24}$$

3.3 Progress Properties

Progress properties are of the form $p \leadsto q$, where $\leadsto$ is the leads-to operator.

Definition 12 ($\leadsto$ operator). *For all state predicates p and q,*

$$[\ (p \leadsto q) \ \equiv\ \mathbf{G}\,(\bullet p \Rightarrow \mathbf{F}\,\bullet q)\ ] \tag{25}$$

In this paper, properties and theorems are often written without explicit quantifications: these are universally quantified over all values of the free variables occurring in them.

Important properties of $\leadsto$ are as follows. For state predicates p, q, and r, we have:

$$[\ (p \Rightarrow q) \ \Rightarrow\ (p \leadsto q)\ ] \tag{Implication}$$

$$[\ (p \leadsto q) \land (q \leadsto r) \ \Rightarrow\ (p \leadsto r)\ ] \tag{Transitivity}$$

$$[\ (p \leadsto q) \ \equiv\ (p \land \sim q \ \leadsto\ q)\ ] \tag{Split-Off-Skip}$$

The main tool for reasoning about progress properties in Unit-B is the *transient operator* **tr**.

Definition 13 (tr operator). *For all state predicate p,* $[\ \mathbf{tr}\,p \ \equiv\ \mathbf{G}\,\mathbf{F}\,;\bullet \sim p\ ]$.

$\mathbf{tr}\,p$ states that state predicate p is infinitely often false. The relationship between **tr** and $\leadsto$ is as follows:

$$p \leadsto \sim p \ =\ \mathbb{1} \leadsto \sim p \ =\ \mathbf{tr}\,p\,. \tag{26}$$

The attractiveness of properties such as $\mathbf{tr}\,p$ is that we can *implement* these using a single event as follows.

Theorem 2 (Implementing tr). *Consider a Unit-B model* $\mathbf{M}$ *and a transient property* $\mathbf{tr}\,p$*. We have* $[\ ex.\mathbf{M} \Rightarrow \mathbf{tr}\,p\ ]$*, if there exists an event*

$$\mathsf{e} \mathrel{\widehat{=}}\ \mathbf{any}\ t\ \mathbf{where}\ g.t.v\ \mathbf{during}\ c.t.v\ \ \mathbf{upon}\ f.t.v\ \ \mathbf{then}\ s.t.v.v'\ \ \mathbf{end}\ ,$$

that is to say $ex.\mathbf{M}$ *entails:*

$$\mathbf{G}\,(\mathbf{G}\ \bullet c \land \mathbf{G}\,\mathbf{F}\,;\bullet f \ \Rightarrow\ \mathbf{F}\,; f\,; act.(\mathsf{e}.t))\ , \tag{LIVE}$$

and parameter t such that $\mathsf{e}.t \in \mathbf{M}$ *and* $ex.\mathbf{M}$ *entails each of the conditions below:*

$$\mathbf{G}\ \bullet(p \Rightarrow c)\ , \qquad \text{(SCH)}$$

$$c \rightsquigarrow f\ , \qquad \text{(PRG)}$$

$$\mathbf{G}\ (\ (p \wedge c \wedge f)\ ;\ act.(\mathsf{e}.t)\ ;\mathbf{true}\ \ \Rightarrow\ \ \mathbf{X}\ ;\bullet\!\sim\! p\ )\ . \qquad \text{(NEG)}$$

Proof. In this case, $\mathbf{G}$ acts as an everywhere operator which allows us to prove $\mathbf{F}; \bullet \sim p$ instead of $\mathbf{G}\ \mathbf{F};\bullet\!\sim\! p$. Additionally, since $[\ \neg s \Rightarrow s\ \equiv\ s\ ]$ for any computation predicate s, we discharge our proof obligation by strengthening $\mathbf{F}\ ;\ \bullet\!\sim\! p$ to its negation, $\mathbf{G}\ \bullet p$.

$$\begin{array}{ll}
 & \mathbf{F}\ ;\bullet\!\sim\! p \\
\Leftarrow & \quad \{\ [\ \mathbf{F}\ ;\mathbf{X} \Rightarrow \mathbf{F}\ ],\ \text{aiming for (NEG)}\ \} \\
 & \mathbf{F}\ ;\mathbf{X}\ ;\bullet\!\sim\! p \\
\Leftarrow & \quad \{\ \text{(NEG)}\ \} \\
 & \mathbf{F}\ ;(p \wedge c \wedge f)\ ;\ act\ ;\mathbf{true} \\
\Leftarrow & \quad \{\ \text{computation calculus}\ \} \\
 & \mathbf{F};f;act;\mathbf{true}\ \wedge\ \mathbf{G}\ \bullet c\ \wedge\ \mathbf{G}\ \bullet p \\
\Leftarrow & \quad \{\ \text{(LIVE); } \mathbf{G}\ \text{is conjunctive}\ \} \\
 & \mathbf{G}\,\mathbf{F};\bullet f\ \wedge\ \mathbf{G}\ \bullet c\ \wedge\ \mathbf{G}\ \bullet p \\
= & \quad \{\ \text{(PRG)}\ \} \\
 & \mathbf{G}\ \bullet c\ \wedge\ \mathbf{G}\ \bullet p \\
= & \quad \{\ \mathbf{G}\ \text{is conjunctive; (SCH)}\ \} \\
 & \mathbf{G}\ \bullet p
\end{array}$$

(Due to space restriction, for the rest of this paper, we only present sketch of proofs of theorems. Detailed proofs are available in [8]).

Condition (SCH) is an invariance property. Condition (PRG) is a progress property. Condition (NEG) states that event $\mathsf{e}.t$ establish $\sim p$ in one step. In practice, often we design c such that it is the same as p and f is $\mathbb{1}$ (i.e., omitting f); as a result, conditions (SCH) and (PRG) are trivial. Condition (NEG) can take into account any invariance property I and can be stated as $[\ (I \wedge p \wedge c \wedge f)\ ;\ act.(\mathsf{e}.t)\ \Rightarrow\ \mathbf{X};\sim\! p\ ]$.

In general, progress properties can be proved using the following *ensure-rule*. The rule relies on proving an unless property and a transient property.

Theorem 3 (The ensure-rule). *For all state predicates p and q,*

$$[\ (p\ \boldsymbol{un}\ q) \wedge (\boldsymbol{tr}\ p \wedge \sim\! q)\ \ \Rightarrow\ \ (p \rightsquigarrow q)\ ] \qquad (27)$$

Proof (Sketch). $p\,\mathbf{un}\,q$ ensures that if p holds then it will hold for infinitely long or eventually q holds. If q holds eventually then we have $p \rightsquigarrow q$. Otherwise, if p holds for infinitely long and $\sim q$ also hold for infinitely long, we have a contradiction, since $\mathbf{tr}\,p \wedge \sim q$ ensures that eventually $p \wedge \sim q$ will be falsified. As a result, if p holds for infinitely long then eventually q has to hold.

3.4 Refinement

In this section, we develop rules for refining Unit-B models such that safety and liveness properties are preserved. Consider a machine $\mathbf{M}$ and a machine $\mathbf{N}$, $\mathbf{N}$ refines $\mathbf{M}$ if

$$[\ ex.\mathbf{N} \Rightarrow ex.\mathbf{M}\ ]\,. \tag{REF}$$

As a result of this definition, any property of $\mathbf{M}$ is also satisfied by $\mathbf{N}$. Similarly to Event-B, refinement is considered in Unit-B on a per event basis. Consider an abstract event $\mathsf{e}.t$ belong to $\mathbf{M}$ and a concrete event $\mathsf{f}.t$ belong to $\mathbf{N}$ as follows.

$$\mathsf{e} \mathrel{\widehat{=}} \textbf{any } t \textbf{ where } g.t.v \textbf{ during } c.t.v \ \textbf{ upon } f.t.v \ \textbf{ then } s.t.v.v' \ \textbf{ end} \tag{28}$$

$$\mathsf{f} \mathrel{\widehat{=}} \textbf{any } t \textbf{ where } h.t.v \textbf{ during } d.t.v \ \textbf{ upon } e.t.v \ \textbf{ then } r.t.v.v' \ \textbf{ end} \tag{29}$$

We have $\mathsf{f}.t$ is a refinement of $\mathsf{e}.t$ if

$$[\ ex.\mathbf{N} \ \Rightarrow\ (act.(f.t) \Rightarrow act.(e.t))\ ]\text{, and} \tag{EVT-SAF}$$

$$[\ ex.\mathbf{N} \ \Rightarrow\ (sched.(f.t) \Rightarrow sched.(e.t))\ ] \tag{EVT-LIVE}$$

A similar rule is presented for the initialisation. The proof that $\mathbf{N}$ refines $\mathbf{M}$ (i.e., (REF)) given conditions such as (EVT-SAF) and (EVT-LIVE) is left out. A special case of event refinement is when the concrete event f is a new event. In this case, f is proved to be a refinement of a special SKIP event which is unscheduled and does not change any abstract variables.

Condition (EVT-SAF) leads to similar proof obligations in Event-B such as *guard strengthening* and *simulation*. We focus here on expanding the condition (EVT-LIVE). The subsequent theorems are related to concrete event f (29) and abstract event e (28), under the assumption that condition (EVT-SAF) has been proved. They illustrate different ways of refining event scheduling information: *weakening the coarse-schedule*, *replacing the coarse-schedule*, *strengthening the fine-schedule*, and *removing the fine-schedule*.

Theorem 4 (Weakening the coarse-schedule). *Given $e = f$. If*

$$[\ ex.\mathbf{N} \quad\Rightarrow\quad \mathbf{G} \bullet (c \Rightarrow d)\ ] \quad \textit{then} \quad [\ ex.\mathbf{N} \Rightarrow (sched.(\mathsf{f}.t) \Rightarrow sched.(\mathsf{e}.t))\ ]\,.$$

Proof (Sketch). The coarse-schedule is at an anti-monotonic position within the definition of $sched$.

Theorem 5 (Replacing the coarse-schedule). *Given $e = f$. If*

$$[\ ex.\mathbf{N} \ \Rightarrow\ c \leadsto d\,] \tag{30}$$

$$[\ ex.\mathbf{N} \ \Rightarrow\ d \ \boldsymbol{un} \ \sim c\,]\,, \tag{31}$$

then $[\ ex.\mathbf{N} \Rightarrow (sched.(\mathsf{f}.t) \Rightarrow sched.(\mathsf{e}.t))\ ]$

Proof (Sketch). Conditions (30) and (31) ensures that if c holds then eventually d holds and it will hold for at least as long as c As a result, if c holds for infinitely long, d also holds for infinitely long. Hence the new schedule ensures that f occurs at least on those cases where e has to occur.

Theorem 6 (Strengthening the fine-schedule). *Given $d = c$. If*

$$[\ ex.\mathbf{N} \quad \Rightarrow \quad \mathbf{G} \bullet (e \Rightarrow f)\ ]\ , \textit{and} \tag{32}$$

$$[\ ex.\mathbf{N} \quad \Rightarrow \quad f \rightsquigarrow e\ ] \tag{33}$$

then $[\ ex.\mathbf{N} \Rightarrow (sched.(\mathsf{f}.t) \Rightarrow sched.(\mathsf{e}.t))\ ]$.

Proof (Sketch). We can prove $sched.(\mathsf{e}.t)$ under the assumptions $sched.(\mathsf{f}.t)$ and $ex.\mathbf{N}$ by calculating from $\mathbf{F}\ ;\ (c \wedge f)\ ;\ act.(\mathsf{e}.t)\ ;\ \mathbf{true}$ (the right hand side of $sched.(\mathsf{e}.t)$) and applying one assumption after the other (in this order (32), (EVT-SAF), $sched.(\mathsf{f}.t)$, (33)) to strengthen it to $\mathbf{G}\ \bullet c \wedge \mathbf{G}\ \mathbf{F}\ ; \bullet f$ (the right hand side of $sched.(\mathsf{e}.t)$).

Theorem 7 (Removing the fine-schedule). *Given $d = c$ and $e = \mathbb{1}$. If*

$$[\ ex.\mathbf{M} \quad \Rightarrow \quad \mathbf{G} \bullet (c \Rightarrow f)\ ] \tag{34}$$

then $[\ ex.\mathbf{N} \Rightarrow (sched.(\mathsf{f}.t) \Rightarrow sched.(\mathsf{e}.t))\ ]$.

Proof (Sketch). Condition (34) ensures that when c holds for infinitely long, f holds for infinitely long, hence we can remove the fine-schedule f, i.e., replaced it by $\mathbb{1}$.

4 Example: A Signal Control System

We illustrate our method by applying it to design a system controlling trains at a station [9]. We first present some informal requirements of the system.

4.1 Requirements

The network at the station contains an *entry block*, several *platform blocks* and an *exiting block*, as seen in Figure 1. Trains arrive on the network at the entry block, then can move into one of the platform blocks before moving to the exiting block and leaving the network. In order to control the trains at the station, signals are positioned at the end of the entry block and each platform block. The train drivers are assumed to obey the signals. The signals are supposed to change from green to red automatically when a train passes by.

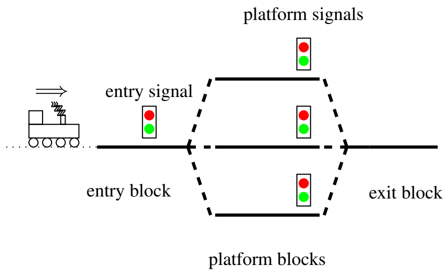


Fig. 1. A signal control system

The most important properties of the system are that (1) there should be no collision between trains (SAF 1), and (2) each train in the network eventually leaves (REQ 2).

SAF 1 There is at most one train on each block

REQ 2 Each train in the network eventually leaves

Refinement strategy. In the initial model, we abstractly model the trains in the network, focusing on REQ 2. In the first refinement, we introduce the topology of the network. We strengthen the model of the system, focusing on SAF 1 in the second refinement. In the third refinement, we introduce the signals and derive a specification for the controller that manages these signals.

4.2 Initial Model

In this initial model, we use a carrier set TRN to denote the set of trains and a variable $trns$ to denote the set of trains currently within the network. Initially $trns$ is assigned the empty set. At this abstract level, we have two events to model a train arriving at the station and a train leaving the station as below.

$$\begin{array}{l}\mathsf{arrive} \mathrel{\widehat{=}} \ \mathbf{any}\ t\ \mathbf{where}\ t \in TRN\ \mathbf{then}\ trns := trns \cup \{t\}\ \ \mathbf{end}\\ \mathsf{depart} \mathrel{\widehat{=}} \ \mathbf{any}\ t\ \mathbf{where}\ t \in TRN\ \mathbf{then}\ trns := trns \setminus \{t\}\ \ \mathbf{end}\end{array}$$

The requirement REQ 2 can be specified as a progress property prg0_1: $t \in trns \leadsto t \notin trns$. According to (26), prg0_1 is equivalent to prg0_2: $\mathbf{tr}\ t \in trns$. In order to implement this transient property, we rely on Theorem 2 and add scheduling information for event depart as follows.

$$\mathsf{depart} \mathrel{\widehat{=}} \ \mathbf{any}\ t\ \mathbf{where}\ t \in TRN\ \mathbf{during}\ t \in trns\ \ \mathbf{then}\ trns := trns \setminus \{t\}\ \ \mathbf{end}$$

Here, we design our depart event to implement the transient property prg0_2 such that conditions (SCH) and (PRG) are trivial. For condition (NEG), it is easy to prove that depart establishes the fact $t \notin trns$ in one step.

Since event arrive will not affect our reasoning about progress properties (it is always unscheduled), we are going to omit the refinement of arrive in the subsequent presentation.

4.3 First Refinement

In this refinement, we first introduce the topology of the network in terms of blocks. We introduce a carrier set BLK and three constants $Entry$, PLF, $Exit$ denoting the entry block, platform blocks and exit block, respectively. A new variable loc is introduced denoting the location of trains in the network, constrained by invariant inv1_1: $loc \in trns \rightarrow BLK$.

For event depart, we strengthen the guard to state that a train can only leave from the exit block. Subsequently, in order to make sure that the schedule is stronger than the guard (condition (SCH-FIS)), we need to strengthen the coarse-schedule accordingly (see Figure 2). In order to prove the refinement of depart, we apply Theorem 5 (coarse-schedule replacing). In particular we need to prove the following conditions:

$$t \in trns \ \leadsto\ t \in trns \land loc.t = Exit \qquad \text{(prg1\_1)}$$

$$t \in trns \land loc.t = Exit\ \ \mathbf{un}\ \ {\sim}(t \in trns) \qquad \text{(un1\_2)}$$

From now on, we focus on reasoning about progress properties, e.g., prg1_1, omitting the reasoning about unless properties, e.g., un1_2. The readers should be convinced

```
depart
any $t$ where
  $t \in trns \land loc.t = Exit$
during
  $t \in trns \land loc.t = Exit$
then
  $trns := trns \setminus \{t\}$
  $loc := \{t\} \lhd loc$
end
```

```
moveout
any $t$ where
  $t \in trns \land loc.t \in PLF$
during
  $t \in trns \land loc.t \in PLF$
then
  $loc.t := Exit$
end
```

```
movein
any $t$ where
  $t \in trns \land loc.t = Entry$
during
  $t \in trns \land loc.t = Entry$
then
  $loc :| \langle \exists p\colon\ p \in PLF\colon\ loc' = loc \Leftarrow \{t \mapsto p\} \rangle$
end
```

Fig. 2. Events of the first refinement

that using Theorem 1, these unless properties are valid for our model. We first apply (Split-Off-Skip) to obtain $t \in trns \land loc.t \neq Exit \;\leadsto\; t \in trns \land loc.t = Exit$ and then apply the transitivity property (Transitivity) of the leads-to operator to establish two progress properties prg1_3 and prg1_4 as follows.

$$t \in trns \land loc.t \neq Exit \;\leadsto\; t \in trns \land loc.t \in PLF \qquad \text{(prg1\_3)}$$

$$t \in trns \land loc.t \in PLF \;\leadsto\; t \in trns \land loc.t = Exit \qquad \text{(prg1\_4)}$$

Consider prg1_4, we first apply the ensure-rule (Theorem 3) to establish two properties (after simplification) as follows:

$$t \in trns \land loc.t \in PLF \;\textbf{un}\; t \in trns \land loc.t = Exit \qquad \text{(un1\_5)}$$

$$\textbf{tr}\, t \in trns \land loc.t \in PLF \qquad \text{(prg1\_6)}$$

We apply Theorem 2 to implement prg1_6 by the new event moveout which has a weakly-fair scheduling (see Figure 2). The proof that moveout implements prg1_6 is straightforward and therefore is omitted.

Similarly, for prg1_3, we apply the ensure-rule and implementing the resulting transient property, i.e., **tr** $t \in trns \land loc.t = Entry$, by event movein in Figure 2.

4.4 Second Refinement

In this refinement, we incorporate the safety requirement stating that there are no collisions between trains within the network, i.e. SAF 1. This is captured by invariant inv2_1 about loc: $\langle \forall\, t_1, t_2\colon\ t_1, t_2 \in trns \land loc.t_1 = loc.t_2\colon\ t_1 = t_2 \rangle$.

The guard of event moveout needs to be strengthened to maintain inv2_1. As a result, we need to modify the schedule information to ensure the *feasibility* condition (SCH-FIS) for Unit-B events stating that the schedules are stronger than the guard. In particular, we add (through strengthening) a fine-schedule to moveout (see Figure 3). The scheduling information for moveout states that for any train t, if t stays in a platform for infinitely long and the exit block becomes free infinitely often, then t can move out of the platform.

We want to highlight the fact that moveout has both coarse- and fine-schedules. In particular, using only either weak- or strong-fairness would be unsatisfactory. Weak-fairness requires for the exit block to be remain free continuously in order for trains to move out. This assumption is not met by the current system. Strong-fairness allows a train to leave if the train is present on the platform intermittently. This assumption is

```
moveout
any $t$ where
  $t \in trns \land loc.t \in PLF \land$
  $Exit \notin \operatorname{ran}.loc$
during
  $t \in trns \land loc.t \in PLF$
upon
  $Exit \notin \operatorname{ran}.loc$
then
  $loc.t := Exit$
end
```

```
movein
any $t$ where
  $t \in trns \land loc.t = Entry \land \langle \exists\, p\colon\ p \in PLF\colon\ p \notin \operatorname{ran}.loc \rangle$
during
  $t \in trns \land loc.t = Entry \land \langle \exists\, p\colon\ p \in PLF\colon\ p \notin \operatorname{ran}.loc \rangle$
then
  $loc :| \langle \exists\, p\colon\ p \in PLF \setminus \operatorname{ran}.loc\colon\ loc' = loc \mathbin{\lhd\!\!\!-} \{t \mapsto p\} \rangle$
end
```

Fig. 3. Events of the second refinement

more flexible than we need since it allows behaviours where a train hops on and off the platform infinitely often. The price of that flexibility is to entangle properties of the exit block with properties of trains: indeed, we would need not only to prove that the train will be on its platform and that the exit block will become free but that both happen simultaneously infinitely often.

We choose to relinquish this flexibility and are therefore capable of structuring our proof better: on one hand, the train stays on its platform as long as necessary; independently, the exit block becomes free infinitely many times.

In order to prove the refinement of moveout, we apply Theorem 6 (fine-schedule strengthening), which requires to prove the following progress property (remember that when an event has no fine schedules, it is assumed to be $\mathbb{1}$).

$$\mathbb{1} \leadsto Exit \notin \operatorname{ran}.loc \qquad \text{(prg2\_3)}$$

Property prg2_3 is equivalent to transient property prg2_4: **tr** $Exit \in \operatorname{ran}.loc$. We satisfy prg2_4 by applying the transient rule (Theorem 2) using event depart where the value for the parameter t is given by $loc^{-1}.Exit$, i.e., the train at the exit block. The proofs of conditions (SCH), (PRG), and (NEG) are straight-forward.

Finally we strengthen the guard of movein and subsequently strengthen its coarse-schedule (see Figure 3). We apply Theorem 5 (coarse-schedule replacing) movein. The detailed proof is omitted here.

4.5 Third Refinement

In this refinement, we introduce the signals associated with different blocks within the network. Variable sgn is introduced to denote the value of the signals associated with different blocks. We focus on the controlling of the platform signals here. In particular, invariants inv3_2 and inv3_3 state that if a platform signal is green (GR) then the exit block is free and the other platform signals are red (RD).

$$\begin{aligned}
&\text{inv3\_1} : sgn \in \{Entry\} \cup PLF \to COLOR \\
&\text{inv3\_2} : \langle \forall\, p\colon\ p \in PLF \land sgn.p = GR\colon\ Exit \notin \operatorname{ran}.loc \rangle \\
&\text{inv3\_3} : \langle \forall\, p, q\colon\ p, q \in PLF \land sgn.p = sgn.q = GR\colon\ p = q \rangle
\end{aligned}$$

We refine the moveout event using the platform signal as shown in Figure 4. The refinement of moveout is justified by applying Theorem 5 (coarse-schedule replacing)

```
moveout
any $t$ where
  $t \in trns \land loc.t \in PLF \land$
  $sgn.(loc.t) = GR$
during
  $t \in trns \land loc.train \in PLF \land$
  $sgn.(loc.t) = GR$
then
  $loc.t := Exit$
  $sgn.(loc.t) := RD$
end
```

```
ctrl_platform
any $p$ where
  $p \in PLF \land p \in \operatorname{ran} .loc \land Exit \notin \operatorname{ran} .loc \land$
  $\langle \forall q\colon\ q \in PLF\colon\ sgn.q = RD \rangle$
during
  $p \in PLF \land p \in \operatorname{ran} .loc \land sgn.p = RD$
upon
  $Exit \notin \operatorname{ran}(loc) \land \langle \forall q\colon\ q \in PLF \land q \neq p\colon\ sgn.q = RD \rangle$
then
  $sgn.p := GR$
end
```

Fig. 4. Events of the third refinement

and Theorem 7 (fine-schedule removing). In particular, replacing the coarse-schedule requires the following transient property

$$\mathbf{tr}\; t \in trns \land loc.t \in PLF \land sgn.(loc.t) = RD\ . \qquad \text{(prg3\_5)}$$

In order to satisfy prg3_5, we introduce a new event ctrl_platform for the controller to change a platform signal to green according to Theorem 2 (see Figure 4). This event ctrl_platform is a specification for the system to control the platform signals preserving both safety and liveness properties of the system. In particular, the scheduling information states that if (1) a platform is occupied and the platform signal is RD infinitely long and (2) the exit block is unoccupied and the other platform signals are all RD infinitely often, then the system should eventually set this platform signal to GR. The refinement of event movein and how the entry signal is controlled is similar and omitted.

5 Conclusion

We presented in this paper Unit-B, a formal method inspired by Event-B and UNITY. Our method allows systems to be developed gradually via refinement and support reasoning about both safety and liveness properties. An important feature of Unit-B is the notion of coarse- and fine-schedules for events. Standard weak- and strong-fairness assumptions can be expressed using these event schedules. We proposed refinement rules to manipulate the coarse- and fine-schedules such that liveness properties are preserved. We illustrated Unit-B by developing a signal control system.

A key observation in Unit-B is the role of event scheduling regarding liveness properties being similar to the role of guards regarding safety properties. Guards prevent events from occurring in some unsafe state so that safety properties will not be violated; similarly, schedules ensure the occurrence of events in order to satisfy liveness properties. Another key aspect of Unit-B is the role of progress properties during refinement. Often, to ensure the validity of a refinement, one needs to prove some progress properties which (eventually) can be implemented (satisfied) by some scheduled events.

Related work. Unit-B and Event-B differ mainly in the scheduling assumptions. In Event-B, event executions are assumed to satisfy the *minimal progress* condition: as long as there are some enabled events, one of them will be executed non-deterministically. Given this assumption, certain liveness properties can be proved for Event-B models such as *progress* and *persistence* [7]. However, this work does not discuss how the refinement notion can be adapted to preserve liveness properties. Moreover, the minimum progress assumption is often either too weak to prove liveness properties or, when it is not, make the proofs needlessly complicated.

TLA+[11] is another formal method based on refinement supporting liveness properties. The execution of a TLA+ model is also captured as a formula with safety and liveness sub-formulae. However, refinement of the liveness part in TLA+ involves calculating explicitly the fairness assumptions of the abstract and concrete models. In our opinion, this is not practical for developing realistic systems. The lack of practical rules for refining the liveness part in TLA+ might stem from the view of the author of TLA+ concerning the *unimportance of liveness* [11, Chapter 8]. In our opinion, liveness properties are as important as safety properties to design correct systems.

Future work. Currently, we only consider superposition refinement in Unit-B where variables are retained during refinement. More generally, variables can be removed and replaced by other variables during refinement (data refinement). We are working on extending Unit-B to provide rules for data refinement.

Another important technique for coping with the difficulties in developing complex systems is composition/decomposition and is already a part of methods such as Event-B and UNITY. We intend to investigate on how this technique can be added to Unit-B, in particular, the role of event scheduling during composition/decomposition.

Given the close relationship between Unit-B and Event-B, we are looking at extending the supporting Rodin platform [2] of Event-B to accomodate Unit-B. We expect to generate the corresponding proof obligations according to different refinement rules such that it can be verified using the existing provers of Rodin.

References

1. Abrial, J.-R.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010)
2. Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. STTT 12(6), 447–466 (2010)
3. Chandy, M., Misra, J.: Parallel program design - a foundation. Addison-Wesley (1989)
4. Dijkstra, E., Scholten, C.: Predicate Calculus and Program Semantics. Springer-Verlag New York, Inc., New York (1990)
5. Dijkstra, R.: Computation calculus: Bridging a formalization gap. Mathematics of Program Construction (January 1998)
6. Dwyer, M., Avrunin, G., Corbett, J.: Patterns in property specifications for finite-state verification. In: ICSE, pp. 411–420 (1999)
7. Hoang, T.S., Abrial, J.-R.: Reasoning about liveness properties in event-B. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 456–471. Springer, Heidelberg (2011)
8. Hudon, S.: A progress preserving refinement. Master's thesis, ETH Zurich (July 2011)
9. Hudon, S., Hoang, T.S.: Development of control systems guided by models of their environment. ENTCS, vol. 280, pp. 57–68 (December 2011)
10. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Software Eng. 3(2), 125–143 (1977)
11. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)

Assume-Guarantee Specifications of State Transition Diagrams for Behavioral Refinement

Christian Prehofer

LMU München and Fraunhofer ESK München,
prehofer@pst.ifi.lmu.de

Abstract. In this paper, we consider extending state transition diagrams (SDs) by new features which add new events, states and transitions. The main goal is to capture when the behavior of a state transition diagram is preserved under such an extension, which we call behavioral refinement. Our behavioral semantics is based on the observable traces of input and output events. We use assume/guarantee specifications to specify both the base SD and the extensions, where assumptions limit the permitted input streams. Based on this, we formalize behavioral refinement and also determine suitable assumptions on the input for the extended SD. We argue that existing techniques for behavioral refinement are limited by only abstracting from newly added events. Instead, we generalize this to new refinement concepts based on the elimination of the added behavior on the trace level. We introduce new refinement relations and show that properties are preserved even when the new features are added.

1 Introduction

State transition diagrams (in short: SD) are used in various forms to model software, e.g. modeling a software component which interacts with the environment based on events. In this paper, we consider behavioral models represented as state transition diagrams which are incrementally extended by new features. The main goal is to reason about the behavior and definedness of such an extended state transition diagram in a modular way.

The idea of *incremental development* is to start with a base model and then to add small features in succession, which add previously unspecified behavior. Extending an SD by a feature means to add new states and transitions.

Assuming such a (syntactic) extension of an SD, the question addressed here is whether the old behavior is preserved when incrementally extending an SD. This we call behavioral refinement. We use a behavioral semantics based on the observable traces of input and output events, respectively. Behavior preservation means that the resulting output trace is unchanged for all input streams, possibly under some abstraction.

As an example consider the lock extension in Figure 1, which adds a new *locked* state and ignores any input in the lock state except for the *unlock* event.

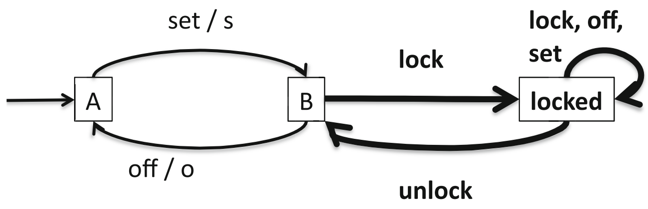


Fig. 1. Locking Feature

By convention, we show the added elements of the new feature in bold text and thicker lines.

In this example, it is easy to see that the behavior of the original base SD is preserved if no *lock* event occurs. While this is a basic compatibility property, we aim to go beyond. As we can see, even after traversing the lock extension, the SD behaves as before. However, during the traversal the externally visible behavior is altered. Furthermore, it may be the case that the SD does not return from the extension if no *unlock* event occurs. We aim to capture these observations in a formal calculus. For this, we address the following two main issues.

The first issue is that the extension also uses the *off* and *set* events of the base SD. Due to this, existing refinement and simulation techniques, e.g. [16, 17, 18, 19, 1, 14] , are not sufficient as they only abstract from newly added events, by definition. This is important in many cases when events are reused in the extension, as in this example above. For this purpose, we use a new concept which eliminates behavior on the trace level. Such an elimination essentially removes the added behavior on the level of observable input/output traces. Technically, we will use entry and exit events to detect and eliminate those segments of the trace that correspond to the newly added behavior.

The second, main point about this example is that we need assumptions on the permitted inputs, both for the base SD as well as for the extension, to reason about the behaviour. For instance, we may want that the extension always terminates and returns to the old SD. In the above example, the extended SD may loop forever in state *locked*. This can be avoided by restrictions on the permitted input. In general, both the base SD and the extension may have assumptions on the permitted input. From these two assumptions, we aim to create a single, combined assumption on the permitted input for the extended SD.

As in other assume/guarantee calculi, we use assumptions to specify what inputs are permitted. These need to make sure that the SD is defined for the permitted inputs, i.e. there is a defined transition for each event in an execution for a given input stream. Our notion of SDs is similar to interface input/output automata in [8], which use a separate state-based model to describe the input assumptions. Here, we use basic predicates to specify the input, not models. Note that our notion of automata is different from interface automata [1, 4], which are intended to specify which input events are permitted for an interface.

For instance, in the example above, we also want to ensure that the *lock* event only occurs in state B, not in state A where it is not defined. Thus, we have to reason about the permitted inputs, both in the extension and in the base SD. This is the main motivation for the assume/guarantee specifications, which are used to formalize this in a modular way.

In summary, the goal is to extend SDs by additional features with new states and transitions, and then to reason about the behavior of the extended SD. For this, we develop a notion of assume/guarantee specifications for SDs. The main idea is to make the permitted inputs and guarantees explicit on the trace level. This follows typical assume/guarantee specifications. We introduce new concepts for semantic refinement based on behavior elimination and present new results when an extension preserves behavior with respect to the base SD.

Our work is similar, on a conceptual level, to aspect- and feature-based programming languages. For these, there are results on so-called conservative extensions or observer aspects, which only add additional behavior but do not modify behavior (see [12, 5]). In other words, we aim to apply these ideas also to SDs, where we are reasoning only about input/output behavior, and not about internal state as on the programming language level. There was recent work to extend automata by aspects, as for instance [20], which includes a calculus for reasoning about automata, but does not identify specific classes of refinement and property preservation. There is earlier work on elimination based refinement in [13], which also permits non-deterministic SDs and also does not require explicit exit events. While there are first results on behavior preservation, in [13] it is not possible to reason about definedness and termination of such extensions.

The paper is organized as follows. In the next section, we introduce the syntax and semantics of SDs. Then, we define syntactic extensions on SDs, followed by behavior eliminations on the semantic level. In Section 3, we introduce new refinement concepts based on these elimination concepts. In Section 4, we present new results to show when a refinement relation can be established for an SD extension. Finally, Section 5 discusses related work, followed by conclusions.

2 State Transition Diagrams

We model software systems by SDs that describe the behavior of a software system. More precisely, an SD consists of

(i) States St, with an initial state $s_0 \in St$
(ii) Input events I and disjoint output events O
(iii) A vector of internal variables v^n, ranging over a vector of values V^n with initial values $V^{initial}$.
(iv) A transition function $tr : St \times I \times V \mapsto St \times V \times O^*$

A transition is triggered by an input event and produces a set of output events. It may have an action that it initiates. This action describes the output events triggered by the transition and the changes on the internal variables. We use the notation *event / action* for transitions. The vector of variables describes the values of the variables and is also called variable valuation.

We focus on deterministic SDs as defined above. For the non-deterministic case, there exist several other issues, as considered in [13].

2.1 Behavioral Semantics

Our semantic model employs an external black-box view of the system. It is based on events from the outside that trigger transitions. Only the observed input and output events are considered, not the internal states. A possible run can be specified by a trace of the events and the resulting output of the SD.

Formally, we assume traces (i, o) over finite and infinite streams over I and O, respectively, denoted as $I^\omega = I^* \cup I^\infty$ and $O^\omega = O^* \cup O^\infty$. Note that for each input event, there is a set of output events if a transition is defined. Hence for the n-th element in i, the n-th element of $S(i)$ is the corresponding set of output events.

For an SD S and a finite or infinite input stream i, we say S is **defined** for i, if there is always a defined transition for each input event in i when executing S with input i. This is written as $Def(S(i))$. For instance, in the above example, the lock SD is undefined for the input *unlock* in the initial state. We write $S(i)$ to denote the output of S for i if S is defined for i.

For a state s and a variable valuation V, we write $S(i, s, V) = (o, s', V')$ to denote the state s' and variable valuation V' after running S at state s with input i and V. This assumes that S is defined for i at state s. We also write $S(i) = (o, s, V)$ if S is run from the initial state. We write $S(i) = s$ if o and V are not of interest. Two SDs are considered equivalent if they behave equivalently for all inputs.

We denote the empty trace as *Nil* and use the following notation on traces:

- $s :: s'$ concatenates two streams, where s is assumed to be finite.
- $a : s$ creates a stream from an element a by appending the stream s.

When clear from the context, we often write just e instead of $\{e\}$ for singleton sets, and also $a : a$ instead of $a : a : Nil$. Furthermore, *first(s)* is the first element of a stream s. We denote by $I \backslash In$ the elimination of elements of *In* from I and by $O + I$ the union of disjoint sets.

As an example, consider the alarm SD as shown in Figure 2, which is extended by a flexible snooze function called Snooze. When the alarm rings, the user can press the snooze button, which sets a new alarm (after a snooze period). In this example, observe that the extension uses new events, like *Snooze*(), but also existing ones like $AskTime()$.

For instance in Figure 2, a possible trace (with input events shown above the corresponding output events) is $T =$

$$\begin{array}{lllllllll}(\mathit{SetAl} & : & \mathit{TimerEvent} & : & \mathit{Snooze} & : & \mathit{TimerEvent} & : & \mathit{AlOff}, \\ \ \mathit{setTimer} & : & \mathit{StartAlarm} & : & \{\mathit{StopAlarm}, \mathit{setTimer}\} & : & \mathit{StartAlarm} & : & \mathit{StopAlarm}).\end{array}$$

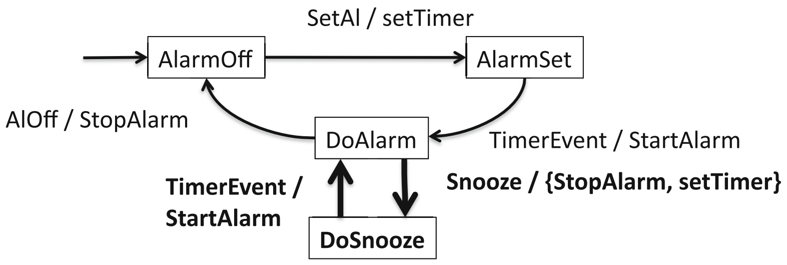


Fig. 2. Alarm extended by Snooze

2.2 Syntactic Extensions of State Transition Diagrams

When adding new features to an SDs, we use the following notion of syntactic extensions of SDs. While we permit any syntactic extensions in the definition below, this will be restricted further below to establish refinement relation on the semantical level.

For an SD S, we say S is **extended** by E to an SD S', if:

(i) S' results from adding both states and transitions on S,
(ii) S' may extend the input and output events of S, and
(iii) S' may add internal variables to S.

S' is also called an **extension** of S by E. Examples of extensions are shown via the bold states and transitions in Figures 1 and 2.

We can alternatively define extensions as a set of states and transitions to be added, which corresponds to a partial or incomplete SD. This is however more subtle, as we need to ensure that a composed SD is well defined, which is implicit here.

3 Assume-Guarantee Specifications and Refinement

In the following, we develop concepts to explicitly specify the assumptions on the input and the resulting output guarantees as in typical assume-guarantee specifications [1]. We will use predicates over finite and infinite streams. We denote assumptions as a predicate A where $A(i)$ is a Boolean value over a stream i, and predicates G as guarantees over a pair of input and output streams, $G(i, o)$.

We use the following notation for assume guarantee specifications over SDs. Assume an SD S, an assumption A and a guarantee G over streams. Then

$$A/S/G$$

states that for all input traces i where A holds, S is defined for i with $S(i) = o$, and $G(i, o)$ holds.

We also write just

$$A/S$$

which then denotes that S is defined for inputs i where $A(i)$ holds.

Assumptions can express two things, unwanted cases and unspecified cases, which we do not distinguish here. Unspecified cases are cases which shall be defined in a later phase by incremental refinement, while unwanted cases must be avoided by the environment and are not allowed.

The typical purpose of assumptions in our treatment of extensions is to specify which inputs are allowed in what phase of a traversal. For instance, when traversing an extension, we may only permit specific events.

The common notion of refinement on SDs is to allow more inputs and to produce less outputs, see for instance [1]. We can formalize more inputs easily by our notion of assumptions. We consider guarantees on the new output based on assumptions for new inputs. Note that we do not allow one to drop individual output events in the output stream.

Assuming a specification $A/S/G$, we can relax the assumptions if the guarantees hold. Also, the guarantees can be strengthened. Formally, $A'/S/G'$ is a **refinement** of $A/S/G$ if $A(i) \implies A'(i)$ and $G'(i,o) \implies G(i,o)$.

3.1 SD Extensions and Refinements

In the following, we aim to cover extensions an SD which add new features with additional behavior. The problem is now that assumptions and guarantees need to consider different input and output events over an extended interface.

For the purpose of refinement, we consider in the following equality on the output traces as follows: Assuming a specification A/S, then A'/S' is a **refinement** of A/S, if $A(i) \implies A'(i)$, and $A(i)$ implies $S(i) = S'(i)$.

This means that S and S' must behave identically for the input permitted for S', i.e. when A holds. In other words, when S' is restricted to the input for A for S, they behave the same. Internally, the two SDs may differ in states and transitions. Compared to the above assume/guarantee specifications the following holds: If A'/S' is a refinement of A/S and G and G' coincide on the inputs permitted for A, then $A'/S/G'$ is a refinement of $A/S/G$.

A typical example is an extension by a new event, after which the system may behave in a completely different way. This notion of refinement is for instance used in [1] when new events are added, but also in [7, 15], even though different formalisms are used. The main limitation here is that no guarantees hold after a new event occurs. In detail, an extension may add new events and the assumptions A do not apply for any input which contains new elements. In the lock example, an assumption predicate over the base SD only considers *set* and *off* as input events, not *lock*. Furthermore, this is only a notion of refinement and does not give any statement when the extended SD is defined.

3.2 Trace Eliminations for Added Features

In the following, we detail our approach to eliminate the behavior of the newly added features. This is used for our notion of behavioral refinement in the following sections. We first discuss important restrictions on SD extensions to determine suitable eliminations and to define the refinement relation. Then, in the subsections below, we define the eliminations and refinement relations precisely.

We determine eliminations of added features on the behavioral level, i.e. on traces. For this, we assume that the new features are triggered by an entry event from I and return to the original SD with an exit event. In the case of the lock feature, the entry event is the *lock* event, and the exit event is the *unlock* event. Then eliminations shall remove all sequences of the form $entryEvent : \cdots : exitEvent$. We call this trace-based eliminations.

For traces over such an extended SD S' over S with extension E, we define eliminations based on entry and exit events. We say that an extension E is **entry-exit triggered**, if there are some **entry events** E_{en}, which do not occur in S. Furthermore, for each entry event $e \in E_{en}$ there is a set of **exit events** $E_{ex,e}$. This means that the states and transitions in E are only reached via some entry transition with an event $e \in E_{en}$. Furthermore, for each such entry point with e, it must be ensured that the extension returns to the original SD S if and only if an event from $E_{ex,e}$ occurs. Furthermore, we assume that it returns to the same state in S where the entry event occurs. This state is also called **join state**, similar to join points in aspect-oriented languages.

We define eliminations based on the entry and exit events as follows. Assuming E, S, and S' as above, an **elimination** el removes all trace segments of the form

$$(i_1 : \cdots : i_n, o_1 : \cdots : o_n),$$

where $i_1 \in E_{en}$, $i_n \in E_{ex,i_1}$ and $i_j \notin E_{ex,i_1}$ for $1 < j < n$. Furthermore, an infinite trace segment $(i_1 :: i, o)$ is eliminated if $i_1 \in E_{en}$ and no element from E_{ex,i_1} occurs in (i, o).

In other words, if the extension does not return, we cut off the complete, infinite part after the entry. Then, we define $el(tr)$ for a trace tr, where the elimination function el is applied from left to right over the full trace tr. This results in a finite trace if there is an entry event without a corresponding exit event.

Note that we use an elimination el in two forms. For input and output traces, we write $el(i, o) = (i', o')$. We also write $el(i) = i'$ which yields an input stream.

As an example, we continue with the above trace T for the alarm SD in Figure 2. The goal is to eliminate the effect of the new Snooze feature. The corresponding traversal through the old SD is

$$\begin{aligned} T' = (\ & SetAl && : TimerEvent && : AlOff, \\ & setTimer && : StartAlarm && : StopAlarm). \end{aligned}$$

In this example, we have eliminated the trace segment *Snooze : TimerEvent* and $\{StopAlarm, setTimer\} : StartAlarm$ which corresponds to the new behavior which the new feature adds. Then, we can show that the original behavior of the SD is preserved by the extended SD "modulo" the elimination.

3.3 Weak Elimination-Based Refinement

We now consider extensions which add behavior temporarily, but then return to existing, old behavior (unless they diverge). For such a case, we use eliminations to define a refinement relation. Elimination is used to compare the input/output traces of the original and the extended SDs. It is a generalization of the typical notions of refinement which remove added behavior by removing the new events. We first define the refinement notion and then discuss its utility.

Assume S over (I, O) is extended to S' with some extension E over (I_E, O_E). The extended system A'/S' is a **weak elimination-based refinement of** A/S, if the following hold:

(i) for any stream i over I, $A(i) \implies A'(i)$ and $S(i) = S'(i)$.
(ii) for any stream i over $I \cup I_E$, if $A'(i)$ holds, then $el(i)$ is a stream over I, $A(el(i))$ and further $el(i, S'(i)) = (el(i), S(el(i)))$.

For this notion of refinement, we require that the extended SD, S', behaves as S under an elimination. We assume that for any permitted input i for S' (i.e. A' holds), the elimination on i results in a syntactically correct input for S and A holds. Otherwise, A may not be defined for $el(i)$. In other words, A' allows more input, even over an extended input event set, but additional traces must correspond to a trace of the original SD. This is enforced by the restriction that $el(i)$ is a stream over I. A possible case when an elimination may not remove all new elements not in I is when an exit event occurs before an entry event.

The notion of weak refinement essentially says that an SD behaves as before unless a new feature is traversed. It will behave as before after multiple use of a new feature, if the new features return to the original SD. Regarding properties of SDs, we can use this notion of refinement to establish safety properties as follows. Safety properties usually state that some "bad" events do not occur. If an SD S does not produce a "bad" output event b under some assumptions A and an extension E also does not produce b (possible under assumptions), then the combined system also does not produce the bad event b. A more basic, but important question is when an extended SD is defined. To establish our refinement we need to fix assumptions under which the extended SD S' is defined, considering both assumptions for the base SD S and the extension E. This will be covered in Section 4.

Compared to the analysis of different kinds of aspects considered in [5], this case is similar to observers with possible non-termination in the extension. In [5], there is also the notion of observers with abortion, i.e. termination of the program. This concept is not sufficient for our setting of SDs as traversals may remain infinitely long in an extension. Instead, we consider possible divergence and termination of the extension by assumptions, as covered in the next section.

3.4 Strong Elimination-Based Refinement

In the above notion of weak elimination-based refinement, we have assumed that the extended SD behaves as the original one under the elimination. We did not require that a traversal through the extension terminates. In case an extension of an SD is entered but the SD does not return from the extension, we only compare the finite parts of the execution. In the following, we define and discuss a stronger notion of refinement, which requires termination for any traversal of the extension.

Assume S over (I, O) is extended to S' with some extension E over (I_E, O_E). The extended system A'/S' is a **strong elimination-based refinement of** A/S if for a stream i over $I \cup I_E$ $A'(i)$ implies the following:

(i) for any stream i over I, $A(i) \implies A'(i)$ and $S(i) = S'(i)$.
(ii) for any stream i over $I \cup I_E$, if $A'(i)$ holds, then $el(i)$ is a stream over I, $A(el(i))$ and further $el(i, S'(i)) = (el(i), S(el(i)))$. Furthermore, if i is infinite, then $el(i)$ is also infinite.

With the last clause in the definition, which is the only difference to the notion of weak refinement, we ensure that a possible extension does not diverge when entered. Clearly, this definition is only sensible if we consider infinite traces which can express divergence.

As strong elimination-based refinement entails weak elimination-based refinement, it can be used to show safety properties as above. As extensions terminate, also many liveness properties are preserved. A typical liveness property is that some (output) event o eventually occurs in all possible executions. In case this holds for the base SD, this is preserved by strong refinement. In this case, an extension may produce extra o events, but it will return to the original SD which eventually produces the o event.

4 Establishing Elimination-Based Refinements

In the following, we aim to establish refinement relations for a given base SD S with an extension E. Based on assumptions for S and E, we show that there exist specific assumptions under which an extension E is an elimination-based refinement. This also serves to reason modularly about extensions of SDs based on properties and assumptions for the SD and the extension. As discussed above, weak elimination-based refinement is suitable for safety properties, while strong elimination-based refinement can also be used for liveness properties.

So far, we have defined assumptions for the inputs of a normal SD. Next we define assumptions specifically for extensions of an SD, which only cover the input events when traversing the added transitions and states in an extension. In other words, we restrict the input while the SD is in the traversal of an extension.

For this purpose, we first generalise assumptions to specific states of an SD. We write $A(s, i)$ for A to hold at some state s with input i. Thus, the above $A(i)$ means that $A(s_o, i)$ for A to hold in the initial state.

Assume S' is an entry-exit triggered extension of S by E. Then for a predicate AE on inputs streams, we denote the **definedness of an extension** E **under** AE as AE/E, to specify that traversals of E in S' are defined. Formally, for all join states s of S', i.e. where an entry event is defined, and some input stream i, where the first element of i is an entry element, and either the last element is the first exit event in i, or no exit event occurs in i, we have: If $AE(i)$ holds, then here is a defined traversal (s, i, o) for some output sequence o. As the extension is entry-exit triggered, the definition entails that E returns to a state in S for any exit event.

Assume S with input events I is extended to S' with some extension E with entry events E_{en} and exit events E_{ex}. For A/S and AE/E, we define the **extension assumptions** $EA(A, E)$ as follows: $EA(A, E)(i)$ holds if

(i) $el(i)$ is a stream over I and $A(el(i))$ holds and
(ii) for any occurrence of an entry event $en \in E_{en}$ in i of the form $i_0 :: en : e :: ex$, where $ex \in E_{ex}$ and e has no exit event, then S' is defined for $i_0 :: en$ and AE holds for $en : e :: ex$.

Intuitively, $EA(A, E)$ has to ensure the following. First, under elimination $EA(A, E)$ has to hold if A holds. Secondly, for the traversal of the new extension, the assumptions for AE have to hold. Note that the first condition, i.e. that $el(i)$ is a trace over I, ensures that new segments in the trace with events not in I are properly started with an entry event and terminated by an exit event. Otherwise, the elimination results in a trace which has events not in I.

We define E to be a **conservative extension** of S if it does not modify variables of S. In other words, the newly added transitions do not modify the variables in S. For conservative extensions, we can show that they do not modify behavior of the extended SD. It may still happen that the control flow does not return from the extension to the original SD.

The following theorem shows that an extended SD is defined for the traces in the extension assumptions $EA(A, E)$.

Theorem 1. *Assume S is extended to S' with some conservative, entry-exit triggered extension E. If A/S and AE/E, then $EA(A, E)/S'$.*

Proof. The proof proceeds by induction on input streams. Assume i is an input of S'. The definedness of S' under $EA(A, E)$ follows from the cases as in the definition of $EA(A, E)$ as follows. In case the input only has elements from I, the case is trivial. In case an entry event occurs in i with S being at a state s, we have to show that S' is defined, which follows from the definition of $EA(A, E)$. Then the traversal in the extension is defined as AE holds for any state and any variable valuation. In case the traversal returns by some exit event to s, we observe that the traversal has not changed the variables of S as it is conservative. Hence the execution of S' at state s continues as S would in this state. As $A(el(i))$ holds, we can infer that also S' is defined until the next occurrence of an entry event. In case the traversal does not return, AE ensures definedness. □

For a property A over streams we say A is **input-consistent**, if $A(i)$ implies $A(i')$ for all prefixes i' of i, i.e. there exists an i'' with $i = i' :: i''$. This assumption is needed for weak refinements, as non-termination may occur in an SD in an extension. Consider an input i which is permitted for the base SD. Then, in an extension an entry event may occur at i', which is a prefix of i. The elimination on the trace of the extended SD will cut off the trace after i' in case of divergence. For the refinement to hold, i' must then also be permitted in the assumptions for the base SD.

Theorem 2. *Assume S is extended to S' with some conservative and entry-exit triggered extension E and A is an input-consistent property. If A/S and AE/E, then $EA(A,E)/S'$ is a weak refinement of S.*

Proof. The proof proceeds by induction on the input stream i of S' and follows the proof of the above theorem. In case E has no entry events, the case is trivial. We show $el(i,S'(i)) = (el(i),S(el(i)))$ inductively over the stream of entry events in i. Assuming it holds for a prefix of i_o of i and $i = i_0 : e_{entry} : i'$. As in the above proof, we relate the execution in S' with S under the elimination. In case a traversal of the extension returns, S' behaves as S as in the proof above after the return, and we can show the equation easily. In case of divergence, there is no exit event in i' and we have $el(i) = i_0$. Then also $el(i,S'(i)) = (el(i),S(el(i))) = (i_0,S(i_0))$. Furthermore, we have to show $A(el(i))$ follows from $A'(i)$. The critical case here is when a trace diverges in an extension, as el will cut off after the last entry event. Here, the assumption on input-consistency is needed to show that $A(el(i))$ holds. □

This theorem shows that there exist assumptions, i.e. $EA(A,E)$, for which an extension (with conditions as above) is a weak elimination-based refinement. Based on this, we can transfer safety properties of an SD to an extended SD as discussed above.

Another issue is to compute $EA(A,E)$ efficiently from the definition of $EA(A,E)$ in practice. The main problem for this is to determine all the input sequences for which a join state can be reached. If this is possible in an SD, we may also compute an effective representation of $EA(A,E)$ by composing input sequences. We illustrate this by the following example.

Consider the lock extension in Figure 1, which adds a new locked state and permits any input in the lock state. Let Set be the base SD, $Lock$ the extension and $SetLock$ be the full SD with the extension as in Figure 1. Then we use regular expressions to define assumption predicates, where the set of input sequences defined by the regular expression defines when the predicate holds.

We define A_{Set} =*(set : off)** and A_{Lock} = *lock : (set,off,lock)* * *:: unlock*. For the extended SD with the lock, we define $A_{SetLock}$ = *(set : (off :set)** *:: (lock : (set,off,lock)* * *:: unlock)** *:: off)** . We use these sets to denote assumptions predicates which hold for all streams in the corresponding set.

In this example, $A_{SetLock}$ is the extension assumption $EA(A_{Set}, A_{Lock})$ for A_{Set}/Set and A_{Lock} based on the above definition. Applying the elimination to $A_{SetLock}$ yields A_{Set} and *set : (off :set)** specifies the inputs leading to the join points (here state B). Based on this, we have $A_{SetLock}/SetLock$ is a weak refinement of A_{Set}/Set.

Notice that we permit that a traversal remains infinitely long in the extension, which we consider as divergence when viewed from a refinement perspective of the base SD. Thus, for $A'_{Lock} = A_{Lock} \cup (lock^{\omega})$, where $lock^{\omega}$ denotes the infinite stream of *lock* events, we still have weak elimination-based refinement assuming $A'_{Lock}/Lock$ (instead of $A_{Lock}/Lock$).

If an extension terminates under specific assumptions, then we can show the stronger property of simulations. For an entry-exit triggered extension E, we say

AE/E terminates if there is not infinite traversal through E which is permitted by AE. For instance, in the lock example, there are possible traversals which stay infinitely long in the extension if no *unlock* event occurs. This can however be disallowed by the assumption AE.

Theorem 3. *Assume S is extended to S' with some conservative and entry-exit triggered extension E. If A/S and AE/E, and further AE/E terminates, then $EA(A,E)/S'$ is a strong refinement of S.*

The proof proceeds as the above result on weak refinement. Here, the proof is easier as all traversals permitted for the extension terminate.

Following the example above, we have $A_{SetLock}/SetLock$ is also a strong elimination-based refinement of A_{Set}/Set with extension $A_{Lock}/Lock$, as all permitted input sequences in A_{Lock} are finite and return to state B.

Recall that strong elimination-based refinement preserves liveness properties. Here, an example is the property that o occurs eventually. This holds, based on strong refinement for $A_{Lock}/Lock$. It does not hold assuming $A'_{Lock}/Lock$, as the traversals may remain infinitely long at the lock state.

5 Related Work

In the following, we discuss related work on statechart refinement and related concepts like UML state machines and other automata models.

Recently, the concept of eliminations was introduced with a focus on compatibility [13], in a setting of non-deterministic SDs with chaos semantics. The approach was also defining eliminations based on traversals of the feature extension, not purely on the trace level as done here. Based on an analysis of the traversals and SD internal states, it was shown when such extensions are a refinement in the sense presented here. Here, we are able to show results on definedness and strong refinement, where the assumptions assure the termination of the extension. This is not possible in prior approaches.

Earlier work on statechart refinement [16][7][15], which is using similar semantic models of statecharts, has developed several rules for refinement. The work in [14] develops a refinement calculus for statecharts as in [16] based on a mapping to the Z language. The basic mechanism for these is also the elimination of new input and output events, as discussed before. Refinement with the focus on stepwise development and composition of services is covered in [3]. Other work on UML in [19], which builds on concepts for object lifecycle modeling [17], considers the problem of consistent inheritance and observation consistency, which are similar to our notion of compatibility. In all of the above, refinement relations are defined by simply removing the new events or ignoring behavior after new events.

For related work on UML modeling, the concepts developed in [18] essentially cover basic cases of refining a state into several ones, which is different and not covered here. The work in [11] focuses on modeling the added features as independent and modular entities, modeled as statechart fragments.

Other work on modularity for model checking [2][10] also considers the problem of extending automata models by new states and transitions. In these works, composition of statecharts leads to proof obligations for specific properties to maintain. These are in turn to be validated by a model checker. Hence, these approaches are quite different from the work presented here. Specifically, they require the specification and establishment of each individual property after the extension. Similar goals have been pursed in the context of aspect-modeling for state machines, as shown in [20].

There is also recent work on compatibility for interface automata [1, 4]. In contrast to interface automata, we model the assumptions of an SD separately. This is conceptually similar to the work on interface input/output automata in [8], which also uses a separate model to describe the input assumptions. The assumptions are modeled as interface automata, which is just a more specific way to denote the input assumptions. However [8] does not focus on refinement and modular reasoning about definedness. More recent work on modal interface automata [9, 6] considers refinement more explicitly by modalities transitions which describe the possible, later refinements. This is different from our work, as we aim at adding behavior without requiring limitations on the SD to be extended.

Compared to our approach of using simply predicates, the work in [8] is using interfaces automata in a more specific way to denote the input assumptions. Unlike [8], we focus more on semantic refinement and modular reasoning about definedness.

6 Conclusions

In this paper, we have presented a new approach to reason about extensions of state transition diagrams based on assume/guarantee specifications. We have focused on extensions which only add new behavior, similar to observer aspects or conservative extensions on a programming language level. A particular problem is that new features may have additional input and output events, but may also reuse existing events. This makes it difficult to reason for what input an extended SD is defined and when it preserves the original behavior. Due to this, existing notions of refinement do not apply. Here, we have developed a new approach towards refinement which allows one to reason about such extended state transition diagrams in a modular way.

In particular, we have developed new refinement concepts for weak and strong refinements, based on an elimination of the newly added behavior on the trace level. These eliminations can be seen as a generalization of typical abstractions, which only remove new input/output events of an extension. Secondly, we have presented an approach for compositional reasoning of such extended SDs using a assume-guarantee calculus. Based on assumptions for the base SD and the extension, we can show when an SD is defined and property preserving after adding the extension. In detail, we show when adding a new feature adds only additional behavior, possibly with divergence. Similar to the considerable work on property preserving aspects and features on the programming language level, we have captured typical extensions like observers also for state transition diagrams in our new approach.

Our approach based on assumptions and guarantees can express various properties of such SDs. We have illustrated that our results can be used to preserve safety and liveness properties when extending an SD. Further work is needed to study in detail how to model and validate typical safety of liveness properties in this form. Also, further work will address how to compute the assumptions needed for an extended SD in an effective way.

Acknowledgements. The author would like to thank Peter Scholz, Martin Wirsing, Sebastian Bauer and Rolf Hennicker for discussion and feedback on this work.

References

[1] Alfaro, L., Henzinger, T.: Interface-based design. In: Broy, M., Grünbauer, J., Harel, D., Hoare, T. (eds.) Engineering Theories of Software Intensive Systems. NATO Science Series, vol. 195, pp. 83–104. Springer, Netherlands (2005)

[2] Blundell, C., Fisler, K., Krishnamurthi, S., Van Hentenrvck, P.: Parameterized interfaces for open system verification of product lines. In: Proceedings of the 19th International Conference on Automated Software Engineering, pp. 258–267 (September 2004)

[3] Broy, M.: Multifunctional software systems: Structured modeling and specification of functional requirements. Sci. Comput. Program. 75, 1193–1214 (2010)

[4] David, A., Larsen, K.G., Legay, A., Nyman, U., Wasowski, A.: Timed I/O automata: a complete specification theory for real-time systems. In: Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2010, pp. 91–100. ACM, New York (2010)

[5] Djoko, S.D., Douence, R., Fradet, P.: Aspects preserving properties. In: Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 2008, pp. 135–145. ACM, New York (2008)

[6] Fischbein, D., Uchitel, S., Braberman, V.: A foundation for behavioural conformance in software product line architectures. In: Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis, ROSATEA 2006, pp. 39–48. ACM, New York (2006)

[7] Klein, C., Prehofer, C., Rumpe, B.: Feature specification and refinement with state transition diagrams. In: Fourth IEEE Workshop on Feature Interactions in Telecommunications Networks and Distributed, pp. 284–297. IOS Press (1997)

[8] Larsen, K.G., Nyman, U., Wąsowski, A.: Interface input/output automata. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 82–97. Springer, Heidelberg (2006)

[9] Larsen, K.G., Nyman, U., Wąsowski, A.: Modal I/O automata for interface and product line theories. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 64–79. Springer, Heidelberg (2007)

[10] Liu, J., Basu, S., Lutz, R.: Compositional model checking of software product lines using variation point obligations. Automated Software Engineering 18, 39–76 (2011)

[11] Prehofer, C.: Plug-and-play composition of features and feature interactions with statechart diagrams. Software and Systems Modeling 3, 221–234 (2004)

[12] Prehofer, C.: Semantic reasoning about feature composition via multiple aspect-weavings. In: Proceedings of the 5th International Conference on Generative Programming and Component Engineering, GPCE 2006, pp. 237–242. ACM, New York (2006)

[13] Prehofer, C.: Behavioral refinement and compatibility of statechart extensions. In: Workshop on Formal Engineering approaches to Software Components and Architectures. I Electronic Notes in Theoretical Computer Science (ENTCS) (2012)

[14] Reeve, G., Reeves, S.: Logic and refinement for charts. In: Proceedings of the 29th Australasian Computer Science Conference, ACSC 2006, vol. 48, pp. 13–23. Australian Computer Society, Inc., Darlinghurst (2006)

[15] Rumpe, B., Klein, C.: Automata describing object behavior. In: Specification of Behavioral Semantics in Object-Oriented Information Modeling, pp. 265–286. Kluwer Academic Publishers (1996)

[16] Scholz, P.: Incremental design of statechart specifications. Science of Computer Programming 40(1), 119–145 (2001)

[17] Schrefl, M., Stumptner, M.: Behavior-consistent specialization of object life cycles. ACM Trans. Softw. Eng. Methodol. 11, 92–148 (2002)

[18] Simons, A.J.H., Stannett, M.P., Bogdanov, K.E., Holcombe, W.M.L.: W.M.l.: Plug and play safely: Rules for behavioural compatibility. In: IProc. 6th IASTED Int. Conf. Software Engineering and Applications, pp. 263–268 (2002)

[19] Stumptner, M., Schrefl, M.: Behavior consistent inheritance in UML. In: Laender, A.H.F., Liddle, S.W., Storey, V.C. (eds.) ER 2000. LNCS, vol. 1920, pp. 527–542. Springer, Heidelberg (2000)

[20] Zhang, G., Hölzl, M.: Hila: High-level aspects for uml state machines. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 104–118. Springer, Heidelberg (2010)

Translating VDM to Alloy

Kenneth Lausdahl

Department of Engineering, Aarhus University
Finlandsgade 24, DK-8200 Aarhus N, Denmark
lausdahl@cs.au.dk

Abstract. The Vienna Development Method is one of the longest established formal methods. Initial software design is often best described using implicit specifications but limited tool support exists to help with the difficult task of validating that such specifications capture their intended meaning. Traditionally, theorem provers are used to prove that specifications are correct but this process is highly dependent on expert users. Alternatively, model finding has proved to be useful for validation of specifications. The Alloy Analyzer is an automated model finder for checking and visualising Alloy specifications. However, to take advantage of the automated analysis of Alloy, the model-oriented VDM specifications must be translated into a constraint-based Alloy specifications. We describe how a subset of VDM can be translated into Alloy and how assertions can be expressed in VDM and checked by the Alloy Analyzer.

1 Introduction

The Vienna Development Method (VDM) [1,2,3] supports modelling and analysis at various levels of abstraction, using a combination of implicit and explicit definitions of functionality, and has a strong record of industrial application [4] for design and specification of software systems. However, one of the limitations of the implicit VDM specifications is the lack of tool support. Existing VDM tools, Overture [5] and VDMTools [6], only provide limited help with the difficult task of validating that an implicit VDM specification captures the intended meaning. The existing tools include standard features like parsers and type checkers, but this only ensures that specifications are correct with regards to syntax and type constraints. The only tool support for semantic validation is a proof obligation generator but this still leaves the difficult task of discharging the proof obligations. Theorem provers such as [7] can be used to discharge the generated proof obligations through an automates translation of VDM to HOL [8,9], but using a theorem prover is usually complicated and requires an expert.

A different approach is to validate the specification through testing by running an interpreter [10]. This is possible for explicit VDM specifications which can be interpreted with actual values. The same approach can be used for implicit specifications through the use of a tool that is able to automatically create explicit definitions of all functions and operations based on their post-conditions, for a subset of the language, and then validate the generated specification through the standard interpreter. This approach has been taken for a subset of VDM that required post-conditions to follow a particular template using conjunctions to separate constraints on the return value [11,12]. While this enables interpretation, it still requires user input, like test cases, whereas an alternative based on model checking requires less or no user input and still provides a larger coverage than testing. Furthermore, such an approach enables easy detection of contradictions in pre- and post-conditions when functions and operations are combined at a system level.

The Alloy Analyzer is a bounded model finder that has proved to be useful for validating specifications in the Alloy language [13]. The analyzer can *find* instances of Alloy specifications, as well as checking user defined assertions. The analyzer can provide immediate visual feedback when an instance is found or present a *core* containing the top level formulas if no instance could be found.

In this paper, we present a translation of VDM to Alloy [13] thereby enabling VDM specifications to be checked by the Alloy Analyzer. This enables users to get immediate feedback both in the form of generated alloy instances, that can be visually displayed and from user-specified properties. We identify a subset of VDM that can be automatically translated and checked in the Alloy Analyzer and give a preserving semantics for the translation, thus identifying where a translation becomes infeasible. Others have already shown that languages like Z, B, Event-B and UML can be translated [14,15,16,17] to Alloy and benefit from the automated analysis the Alloy Analyzer provides. The VDM language does not contain any direct way to capture system properties, also called validation conjectures [18, p. 191], which is equivalent to the assertions used in model checking but we show a new way of using existing VDM expressions to express such properties.

The structure of this paper is as follows. Section 2 describes the languages VDM and Alloy and how they relate. Section 3 formally defines a subset of VDM and defines the translation rules and the limitations of the translation. Section 4 describes how VDM specifications can be checked using this translation. Section 5 describes the findings discovered by applying the translation to a number of VDM specifications. Finally, Section 7 discusses the contribution of this paper.

2 VDM Models and Alloy Instances

The Vienna Development Method is one of the longest established model-oriented formal methods, and was originally developed at the IBM laboratories in Vienna in the 1970's. The VDM Specification Language is a higher-order language which is standardised by the International Organization for Standardization (ISO), and has a formally defined syntax, and both static and dynamic semantics [19,20]. The VDM language employs a three valued logic; values may be true, false or bottom (undefined), using Logic of Partial Functions (LPF) where the order of the operands is unimportant[21][1]. Models in VDM are based on data type definitions built from simple abstract types using booleans, natural numbers, characters and type constructors for record, product, union, map, (finite) set and sequences. Type membership may be restricted by predicate invariants. Persistent state is defined by means of typed variables, again restricted by invariants. Operations that may modify the state can be defined implicitly, using standard

[1] The existing VDM interpreter is, however, depended on the operand order.

pre- and post-condition predicates, or explicitly, using imperative statements. Such operations denote relations between inputs, outputs and states before and after execution. Functions are defined in a similar way to operations, but may not refer to state variables.

The traditional approach is to start at a high abstraction level using implicit VDM definitions [2] and then refine the model into an explicit executable model. This approach enables validation of the abstract design before a lot of detail is added. However, there is only limited tool support for validating such implicit VDM specifications in contrast to explicit specifications which can be interpreted. The only semantic validation supported is generation of proof obligations that have to be proven manually or with the help of a theorem prover.

The following is a subset of a VDM specification of a telephone system [22] later published as both a Z and B specification [23,24]:

module *telephone*

$Subscriber = $ **token**

$Initiator = \mathrm{AI} \mid \mathrm{WI} \mid \mathrm{SI}$

$Recipient = \mathrm{WR} \mid \mathrm{SR}$

$Status = \mathrm{FR} \mid \mathrm{UN} \mid Initiator \mid Recipient$

$$\textbf{state}\ Exchange :: \begin{array}{rl} status & : Subscriber \xrightarrow{m} Status \\ calls & : Subscriber \xleftrightarrow{m} Subscriber \end{array}$$

$$\begin{array}{l}\textbf{inv}\ (mk\text{-}Exchange(s,c)) \triangleq \forall i \in \textbf{dom}\, c \cdot \\ \qquad (s(i) = \mathrm{WI} \wedge s(c(i)) = \mathrm{WR}) \vee \\ \qquad (s(i) = \mathrm{SI} \wedge s(c(i)) = \mathrm{SR})\end{array}$$

$Lift\ (s\colon Subscriber)$

ext wr *status*

pre $s \in \textbf{dom}\,(status \rhd \{\mathrm{FR}\})$

post $status = \overleftarrow{status} \dagger \{s \to \mathrm{AI}\}$

$ClearSpeak\ (i\colon Subscriber)$

ext wr *status*

  wr *calls*

pre $s \in \textbf{dom}\,(status \rhd \{\mathrm{SI}\})$

post $status = \overleftarrow{status} \dagger \{i \to \mathrm{FR}, calls(i) \to \mathrm{UN}\} \wedge calls = \{i\} \mathbin{⩤} \overleftarrow{calls}$

end *telephone*

The specification describes a telephone system in terms of communication between *Subscriber*s and controlled by a mapping from *Subscriber* to *Status*. The statuses are free (FR), Unavailable (UN), Attempting- (AI), Waiting- (WI) and Speaking-initiator (SI) and Waiting- (WR) and Speaking-recipient (SR). The *Lift* function only accepts subscribers that have the FR status and requires the state after to overwrite the status to AI for the given subscriber, it gives a frame condition stating that the only part of the state that may be changed or read is $status$. The $ClearSpeak$ operation only accepts a subscriber with the status SI and requires the status of that subscriber to be set to FR and the subscriber it was in a conversation with, represented by $calls(i)$, to have the status UN. It also requires that no call exists in the calls map for the given subscriber.

Alloy [13] is a declarative formal specification language for describing software abstractions. Alloy enables fully automatic analysis that gives immediate feedback but, unlike theorem proving, the analysis is not complete, and it only examines a finite space of cases, which, due to recent advances in constraint-solving technology, usually is often vast and therefore offers a degree of coverage unattainable by testing. At the core, Alloy is based on relations over atoms with a logic that is small, simple and expressive. It is based on a relational logic that combines the quantifiers of first-order logic with operators of relational calculus and easy to learn and understand if one already is familiar with basic set theory. The Alloy language is more than just logic; it provides ways to organise a model, build larger models based on smaller ones and a way to factor out components for reuse. The language also provides a number of shorthand and declarations needed to communicate with the Alloy Analyzer. And, finally, the language includes modules, polymorphism, parametrized functions etc. but some features are unique to Alloy including *signatures* and the notion of *scope*. Alloy modules consists of:

- *Module header*: Header identifying the module, enabling them to be opened and reused by other modules.
- *Signature definitions*: Each signature, labelled **sig**, represents a set of atoms and may introduce fields that each represents a relation that relates atoms.
- *Constraint paragraphs*: Various forms of constraints and expressions, labelled **fact**, **fun** and **pred**.
- *Assertions*: Records properties that are expected to hold and labelled **assert**.
- *Commands*: Instructions to the analyzer to perform a particular analysis and labelled **run** or **check**.

The following example illustrates an Alloy specification of the Telephone example. It starts by defining a number of signatures $TOKEN$, $Subscriber, \cdots$. Then two predicates are specified including a *run* command. Predicates are just normal formulas, and *run* commands are used to ask the Alloy Analyzer to find a satisfying assignment with a given scope. The *run* command given in this example asks the Alloy Analyzer to find a satisfying assigment with the default number of instances set to 3 and with two additional constraints restricting the number of instances of Exchange to at most 2 and Subscriber to at most 1 instance. The result can then be graphically represented as shown in figure 1.

```
module telephone
open util/relation
sig TOKEN{}
sig Subscriber extends TOKEN{}
one sig AI, SI, WI,SR, WR,FR, UN{}
sig Initiator in AI + SI + WI{}
sig Recipient in SR + WR{}
sig Status in FR + UN + Initiator + Recipient{}
sig Exchange{
    status: Subscriber →lone Status,
    calls: Subscriber lone →lone Subscriber
}{ functional[status,Exchange] and
   injective[calls,Exchange] and functional[calls,Exchange] and
     all i : dom[calls] | ( status[i] = WI and status[calls[i]]= WR) or
                          ( status[i] = SI and status[calls[i]] = SR)
}

pred Lift(e : Exchange, e’ : Exchange, s: Subscriber)
{ e’·calls = e·calls
    s in dom[e·status :>FR]
    e’·status = e·status ++ s →AI
}
run Lift for 3 but 2 Exchange, 1 Subscriber

pred ClearSpeak(e : Exchange, e’ : Exchange, i: Subscriber)
{ i in dom[e·status :>SI]
    e’·status = e·status ++ (i →FR + e·calls[i] →UN)
    e’·calls = univ -i <:e·calls
    i in dom[e·calls]
}
```

The instance presented in figure 1 shows two *Exchange* objects with a 0 and 1 index and a label indicating which argument of *Lift* they represent. There are no indexes given for the *Subscriber*, *AI* and *FR* since only a single object of each exists. The arrows represent the relation *Status*, and the brackets indicate how respectively *AI* and *FR* are related to the two exchange objects. It is important to notice that Alloy does not differentiate between pre- and post-state like VDM; instead it is up to the modeller to construct e.g. a predicate such that the pre- and post-state is represented which in this case is done by two additional arguments e and e'.

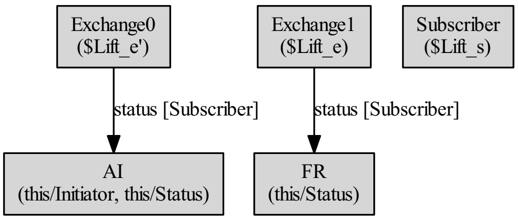


Fig. 1. An instance found by running the Alloy Analyzer on the Telephone with the lift predicate

3 Semantics Preserving Translation

Software abstractions are expressible in both VDM and Alloy while their foundation is fundamentally different, with VDM being a higher-order, model-oriented language and Alloy a first-order constraint bases language built on relations of atoms which are primitive entities that are indivisible, immutable and uninterpreted. The basic building blocks of VDM are type constructs that are used to capture system entities, and can be arbitrary complex using cyclic dependencies and nesting through e.g. sets, sequences, maps or records. Types can furthermore be annotated with invariants to record properties that must always hold between entities or entity fields. An Alloy signature is similar to a VDM type but limited to only expressing relations between atoms and thus not able to directly express e.g. sets of sets often used in VDM. However, by introducing extra atoms and relations most VDM types can be translated including constraints defined by invariants.

A translation from VDM to Alloy cannot always be done faithfully because VDM is more expressive then Alloy. However, despite the large proportion of VDM specifications that are written with higher-order constructs most of them are not fundamentally higher-order and can be expressed in a first-order way. If a VDM specification only uses the definitions and types shown below, and the expressions presented in detail later, then it can be translated into a semantically equivalent Alloy specification. The translation is specified for a subset of VDM that excludes recursive functions and is restricted to a two-value semantics of VDM. The subset shown below ensures that a VDM specification can be directly represented in Alloy. The language consists of *Modules* which are directly translatable to an Alloy module and thus not shown here. A module is a container that contains a number of *Definition*s.

$$\mathit{Definition} = \mathit{TypeDef} \mid \mathit{StateDef} \mid \mathit{OpDef} \mid \mathit{FunDef} \mid \mathit{ValueDef}$$

$$\mathit{TypeDef} = \mathit{Id} \times \mathit{Type}$$

$$\mathit{Type} = \mathit{BasicType} \mid \mathit{ConstructiveType}$$

$$\begin{aligned}\mathit{ConstructiveType} = {} & \mathit{InMapType} \mid \mathit{MapType} \mid \mathit{NamedInvType} \mid \mathit{ProductType} \mid \\ & \mathit{SeqType} \mid \mathit{SetType} \mid \mathit{UnionType}\end{aligned}$$

$$\begin{array}{lrl}\mathit{NamedInvType} :: & \mathit{type} : & \mathit{NamedType} \mid \mathit{RecordType} \\ & \mathit{invpattern} : & \mathit{Pattern} \\ & \mathit{inv} : & \mathit{Exp}\end{array}$$

TypeDef is a name type pair with a *Type* that either is a *BasicType* (*Token*, *Quote*, *Int*), or a *ConstructiveType*. Basic types translates easily to Alloy, shown here with the abstract VDM representation on the left and the Alloy syntax on the right where $[\![ ]\!]$ denotes the meaning of the VDM component:

$$\begin{aligned}[\![\mathit{TokenType}]\!] &= \texttt{\textbf{sig}}\ \mathit{Token}\{\} \\ [\![\mathit{QuoteType}(\mathit{tag})]\!] &= \texttt{\textbf{one sig}}\ \texttt{tag}\{\}\end{aligned}$$

Tokens translate directly to signatures with no fields where a quote type translates to a *one* signature indicating that only a single instance can ever exist in the universe during analysis. Signatures in Alloy can be used to express sub-typing relations. This enables translation of named types in VDM, which are either record types or a named composite type; the translation of $NamedType$ is based on the enclosing type:

$$\begin{aligned}[\![NamedType(name, t)]\!] = \; & (t \in BasicType - IntType \vee \\ & t \in ConstructedType - UnionType \\ & \;\Rightarrow \textbf{sig}\ name\ \textbf{extends}\ [\![t]\!]\{\}) \\ & \wedge(t = IntType \\ & \;\Rightarrow \textbf{sig}\ name\ \textbf{in}\ \textbf{Int}\{\}) \\ & \wedge(t = UnionType(t1, t2) \\ & \;\Rightarrow \textbf{sig}\ name\ \textbf{in}\ [\![T1]\!] + [\![T2]\!]\{\})\end{aligned}$$

If the enclosing type of the $NamedType$ is integer or union then a new signature is created using the Alloy in notation to indicate that the signature is a sub-signature of another signature of a union of signatures. The symbol $-$ is used for set difference restricting the types and $+$ for set union creating a union of types in Alloy. The VDM invariant type adds an invariant to a $NamedType$ in the form of a boolean expression that must hold for all instances of that type. This is representable by an Alloy fact using a for-all expression quantifying over all objects that belong to the type:

$$[\![NamedInvType\begin{pmatrix}NamedType(name, type),\\ pattern, exp\end{pmatrix}]\!] = \begin{array}{l}[\![NamedType(name, type)]\!] \\ \textbf{fact}\ \texttt{name}Inv\{ \\ \textbf{all}\ [\![pattern]\!]\texttt{:name} \mid [\![exp]\!]\}\end{array}$$

The behaviour is modelled around state in VDM, through operations and functions. Functions frequently use recursion to manipulate sets, sequences or other data structures. Common to operations and functions is that they both declare pre- and post-conditions which are essentially predicates. However, Alloy does not include any notion of state and therefore additional arguments must be added to enable operations to refer to pre- and post-states. The VDM state is easily representable by an Alloy signature:

$$[\![StateDefinition\begin{pmatrix}name,\\ fields,\\ inv\end{pmatrix})]\!] = \begin{array}{l}\textbf{let}\ relations = \{f\colon [\![fields(f)]\!]\texttt{-}t \mid f \in \textbf{dom}\, fields\} \\ \textbf{in sig}\ name\{relations\} \\ \quad \{getInvs(fields)\ \textbf{and}\ [\![inv]\!]\}\end{array}$$

note that unlike VDM, multiple instances can exist of this signature, multiple instances are needed to present the pre- and post-state used for each operation invocation as opposed to VDM, where the pre-, post-states are implicitly handled and accessed through the state field identifiers or old identifiers ($\overleftarrow{id}$) as used in post-conditions. The function *getInvs* is a utility function that adds the necessary constraints to a signature when any of the fields represents a mapping in the VDM specification.

```
$getInvs\colon (Id \xrightarrow{m} Type) \to AlloyExp$-set
$getInvs(fields) invs ==$
let $f \in$ dom $fields$ in
cases $fields(f)$ of
  $Map \to (\{\texttt{functional}[f])\} \cup getInvs(\{f\} <\text{-:} fields)$
$InMap \to (\{\texttt{functional}[f]\} \cup \texttt{injective}[f]) \wedge getInvs(\{f\} <\text{-:} fields)$
others { }
end
```

Functions: Implicit functions, only consisting of pre- and post-conditions, can be represented by Alloy predicates. Unless the return type is boolean, an additional argument must be added to the predicate with the type and pattern. Explicit functions that additionally declare a body resemble functions in Alloy. However, a direct translation only preserves the semantics if the VDM function does not use or return any constructive expressions (mk-). The VDM semantics creates a new fresh object based on the arguments of a mk-expression; whereas Alloy search the universe to find a match. This can be illustrated by a VDM function returning a record $Person$ with two fields $firstname$ and $lastname$ with a body defined as $mk\text{-}Person(first, last)$ which always returns a person record. In Alloy, the function body could be written like $\{p\colon Person \mid p.firstname = first$ **and** $p.lastname = last\}$. This, however, may return either a person or an empty set in the case where the universe does not include a person that matches the one requested. There are two solutions in these kind of cases: *a*) create a generator fact that pre-populates the universe with all instances that may ever be needed in the specification, or *b*) use a predicate with an additional argument representing the return value. The advantage with the latter solution is that it does not create instances that may not be needed and thus slowing down the analysis; it also avoids the difficult task of calculating exactly which instance is needed. An example of the latter is illustrated below, showing how a VDM function returning a record, can be expressed with a predicate in Alloy; note the extra argument p representing the return value:

```
$getPerson\colon Id \times Id \to Person$          pred getPerson[first,last : Id, p : Person]{
$getPerson(first, last)p ==$                   p·firstname = first and
  $mk\text{-}Person(first, last)$                  p·lastname = last}
```

Operations: Implicit operations are like implicit functions with the exception that pre- and post-condition expressions are allowed to refer to state fields. Pre-conditions are only allowed to refer pre-state by state field names. Post-conditions are allowed to refer to pre-state by old state field names ($\overleftarrow{id}$ and to post-state by state field names. Like implicit functions, a predicate is used to represent the operation but two additional arguments are required to encode the notion of pre- and post-state. This is done by adding arguments e.g. $e, e'\colon Exchange$ where we define the first e to be the pre-state and e' the post-state. The pre-condition expression may have free variable references, that, in VDM, will be bound to the state fields, but instead they must be joined with the pre-state instance e in this case. The same applies for post-conditions with the exception that names without the old symbol must be joined to the post-state (e') and old names (e) to the pre-state as illustrated:

```
$[\![ImplOpDef(name, args, retId \times type, pre, post, frame)]\!] =$
$\textbf{let}\ StateDef(stateName, fields, \text{-}) = getGlobalState(),$
    $args' = addPrePostArgs(getArguments(args), stateName),$
    $pre' = updateStateIds(pre, fields, getPreName()),$
    $post' = updateStateIds(post, fields, getPostName()),$
    $post'' = updateOldStateIds(post', fields, getPreName()),$
    $frame' = getExtConstraints(frame)$
    $\textbf{in}$
      $(type \in \mathbb{B} \Rightarrow \texttt{pred}\ name(args')\{[\![pre']\!] \wedge [\![post'']\!] \wedge frame'\})$
      $\wedge(type \notin \mathbb{B} \Rightarrow \textbf{let}\ args'' = appendArg(args', retId, type)\ \textbf{in}$
            $\texttt{pred}\ name(args'')\{[\![pre']\!] \wedge [\![post'']\!] \wedge frame'\})$
```

Operations have implicit assumptions about when state is changed which is connected to frame-conditions. If no frame-condition is given for a state identifier or a read clause is specified then that identifier must remain constant and requires the addition of an equality of post-state and pre-identifier identifier e.g. $e'.calls = e.calls$.

Expressions: Semantically most the VDM expressions included in this work have equivalent expressions in Alloy but use a different concrete syntax. A subset of VDM expressions has been chosen based on the case studies made and is only a subset of the expressions which can be translated to Alloy:

$$\begin{aligned}&Exp = \\ &\quad ApplyExp \mid BinExp \mid ExistsExp \mid FieldExp \mid ForallExp \mid IfExp \mid LetExp \mid \\ &\quad MapEnumExp \mid MapExp \mid MkExp \mid MkType \mid QuoteExp \mid SeqCompExp \mid \\ &\quad SetCompExp \mid SetEnumExp \mid TupleExp \mid UnaryExp \mid VarExp\end{aligned}$$

where unary expressions includes the operators: $dom, dunion, inverse, not, rng$ and the binary operators are: $\wedge, \vee, =, \neq, \Rightarrow, \in, \dagger, \cup, \rhd, \lhd, \cap$. The variable expression has a special case where the variable identifier refers to a set of sets that is encoded as a set of a signature with a field representing the second set. Thus, the inner set must be returned if such an implicit signature is referred:

$$\begin{aligned}[\![VarExp(id, type)]\!]\_var = \ & type = SetType(SetType(\text{-})) \Rightarrow id.x \\ & type \neq SetType(SetType(\text{-})) \Rightarrow id\end{aligned}$$

The semantics of relational join in Alloy differs from that of map application in VDM. The difference is how application of keys outside the domain of a map is handled: in VDM this results in an error whereas, in Alloy, it evaluates to an empty set because of the join. The solution is to add a constraint to the join requiring the key to exist in the domain of the map.

$$[\![ApplyExp(root, i)]\!] = [\![i]\!]\ \texttt{in}\ ([\![root]\!]).\texttt{univ and}\ [\![root]\!][[\![i]\!]]$$

$$\begin{aligned}&[\![ForallExp(bind, exp)]\!] = \\ &\quad getFreeVars(bind) = \{\,\} \Rightarrow \texttt{all}\ [\![bind]\!]\_bind \mid [\![exp]\!] \\ &\quad getFreeVars(bind) \neq \{\,\} \Rightarrow \\ &\qquad \texttt{all}\ getTypeBinds(bind) \mid [\![bind]\!]\_in\ \texttt{implies}\ [\![exp]\!]\end{aligned}$$

$$\begin{array}{lll}
[\![\mathit{UnaryExp}(\mathit{exp}, \mathbf{dunion})]\!] = \textbf{let } t = \mathit{typeof}(\mathit{exp}) \textbf{ in} & & \\
\quad t = \mathit{SetType}(\mathit{RecordType}(\text{-})) & \Rightarrow & \texttt{toSet[}[\![\mathit{exp}]\!]\texttt{]} \\
\quad t = \mathit{NamedInvType}(\mathit{NamedType}(\mathit{name}, \mathit{SetType}(\text{-})), \text{-}, \text{-}) & \Rightarrow & [\![\mathit{name}]\!].x
\end{array}$$

3.1 Limitations

The semantics preserving translation given here does not cover undefindeness in VDM or lambda functions and there are a few limitations to how VDM specifications may be written and how the scope needs to be given while using the Alloy Analyzer.

Identifiers. The Alloy language treats identifiers differently from VDM, and as a result of this the translation requires all argument identifiers for functions and predicates to be disjoint from any field names used in signatures. If this is violated, the type checker in Alloy cannot detect the correct type when the join operator is used, and this results in type errors.

Scope — sequences. Sequences are an essential part of VDM, but sequences are not directly representable in Alloy. However, the Alloy Analyzer includes a standard library that represents a sequence as a relation $int\text{-} > univ$ mapping a number to an instance and posing an ordering on the integers used. This solutions works well for VDM sequences but is hard to use since the scope used for the integers used in the domain decides the maximal length of any sequence, and if exceeded, the sequence is silently truncated. Moreover the only way the scope can be changed for the integers used as index in sequences is by changing the default scope used by a command.

Values. The VDM values are a constant representation of an object of any type that is used in specifications to compare calculated results against. While this works well for all types in VDM it quickly makes analysis in Alloy unnecessary slow due to the increased scope required. The values can be split into two categories: basic typed values and values using constructive types. Basic typed values can be represented in Alloy as *one* signatures and has only a small impact on the performance while the constructed types requires a generator fact to populate the complex structure, essentially forcing all required instances to exist. A cleaner and more abstract way to represent these values are VDM functions, which can be represented by Alloy predicates.

The translation does not include recursion or statements used in operation bodies because of limitations of the Alloy Analyzer. Both recursion and a subset of the VDM statements are expressible in Alloy but the analyzer will only check recursive functions for a depth of three which rarely is enough for most VDM specifications; thus the benefit of a translation is limited. A similar problem exists if e.g. a for-statement loops over a sequence and calculating whether or not a certain condition is reached for termination. In this case the loop can be unfolded in Alloy to a predefined number of loops. However, this dramatically slows down the analysis to hours instead of seconds.

4 Checking Implicit VDM Specifications

The translation from VDM into Alloy extends the existing syntax and type checking with the features of the Alloy Analyzer, which, unlike the VDM proof obligation generator, is fully automated and do not require any human intervention. The two main features are:

1. *Simulation*: Finding instances of state or execution that satisfies certain properties.
2. *Checking*: Finding counterexamples — instances that violates a given property.

Simulation is similar to interpretation of explicit specifications, but instead of actual values, symbolic values are used, which are limited to the size of the scope. The result of a simulation is any instance that respects the constraints (predicates, facts) of the specification and thus non-deterministicly decided. If an instance is found, then its structure can be graphically visualized, and examined through an evaluator; however, if no instance is found, a core is given. A core is a collection of possibly contradicting formulas that describe what prevents an instance from being found. If the core is consistent, then there may just not exist any instance within the specified scope and thus a larger scope may be needed. Enabling simulation of implicit specifications is a simple way to verify that a desired instance can be obtained, which turned out not to be the case for two of the specifications translated during this work. Both specifications, Hotel [25,13] and Telephone [22], had problems with the use of pre- and post-state references and thus used a post-state where a pre-state should have been used resulting in a core instead of an instance. Example of a post-condition in the Telephone example:

$$\begin{aligned}&status = \overleftharpoon{status} \dagger \{calls(i) \to \text{FR}\} \wedge\\&calls = \{i\} \lhd \overleftharpoon{calls}\end{aligned}$$

where i is applies to the map $calls$ while the second line states that the domain of $calls$ are restricted by i and this i cannot be in the domain.

While simulation searches for an instance that satisfies given properties it does not guarantee that only instances that respect the properties exist. Checking is another technique where a model finder tries to find an instance which respects the specification but violates a given property defined as an assertion. Model finding is more efficient than testing, since the user only has to specify an assertion that has to always hold and the model finder will search for an instance which violates this. The VDM language does not contain any official way to capture assertions; however, the notion of *validation conjectures* [18, p. 191] is mentioned in relation with proof techniques but suggested to be written as part of the model documentation. Thus a high level informal assertion for the telephone example may look like this:

$$Lift \to Connect \to ClearWait$$

One way to interpret this informal assertion is that there is some relation between the states and that the state before *Lift* and after *ClearWait* are equal. A conservative translation of this can be expressed as follows as an Alloy assertion:

```
assert liftConnectClearWait{
    all e,e',e'',e''' : Exchange, s1,s2 : Subscriber |
        (free[e,s1] and Lift[e,e',s1] and
        Connect[e',e'',s1,s2] and
        ClearWait[e'',e''',s1]) implies eq[e,e''']}
```

where *free* and *eq* are utility functions that respectively sets up the initial state and compare the final and first state. If an counterexample is found then the Alloy Analyzer is able to visually display it like for any simulation.

Alternatively to expressing assertions over VDM specifications in different notations, we propose to use the operation quotation [2] in VDM to express such assertions in the VDM language. For each function or operation in VDM, two implicit boolean functions *pre-* and *post-* representing the pre- and post-condition, exist. By utilizing this, the above assertion can be written in the VDM notation as follows:

$$\begin{aligned}
&\forall e, e', e'', e''' \colon \mathit{Exchange}, s1, s2 \colon \mathit{Subscriber} \cdot \\
&\quad (\mathit{free}(e, s1) \wedge \\
&\quad \mathit{pre\text{-}Lift}(s1, e) \wedge \mathit{post\text{-}Lift}(s1, e, e') \wedge \\
&\quad \mathit{pre\text{-}Connect}(s1, s2, e') \wedge \mathit{post\text{-}Connect}(s1, s2, e'') \wedge \\
&\quad \mathit{pre\text{-}ClearWait}(s1, e'') \wedge \mathit{post\text{-}ClearWait}(s1, e'', e''')) \;\Rightarrow\; \mathit{eq}(e, e''')
\end{aligned}$$

the *free* and *eq* functions are equal to the one used in Alloy and just represented as simple boolean functions in VDM. The *pre-* functions take the same arguments as the function they are guarding with the addition of the pre-state. The same applies for the *post-*functions with the addition of the pre- and post-state. The assertion is conservative in the sense that it only has to hold in the case, where all functions denoted by pre- and post- are true, which does not allow cases where a post-condition is false but still implies a valid pre-condition.

5 Case Studies

The translation has been applied to a number of implicit VDM specifications. The intended outcome was to validate if the specifications could be represented in Alloy, and to check if any errors could be found in the specifications. The translation rules have been incorporated in a prototype tool that outputs an Alloy specification for the supported subset of VDM. The four most interesting specifications that have been analysed are:

Telephone. The Telephone example [22] shown in Section 2 was originally written in VDM [26,21] and later translated into Z [23] and B [24]. The analysis revealed that two post-conditions contained contradictions, and thus resulted in a core instead of an instance. The original post-condition of the *ClearSpeak* operation is defined as:

$$\mathit{ClearSpeak}\ (i \colon \mathit{Subscriber})$$

$$\textbf{post}\ \mathit{status} = \overleftarrow{\mathit{status}} \dagger \{i \to \mathrm{FR}, \mathit{calls}(i) \to \mathrm{UN}\} \wedge \mathit{calls} = \{i\} \mathrel{⩤} \overleftarrow{\mathit{calls}}$$

This is represented in Alloy as shown below where lines 2 and 3 represent the VDM post-condition shown above, and line 4 the extra constrains required by VDM map application:

```
1  pred ClearSpeak(e : Exchange, e' : Exchange, i: Subscriber){
2    e'·status = e·status ++ (i→ FR + e'·calls[i] →UN)
3    e'·calls = (univ-i) <:e·calls
4    i in dom[e'·calls]}
```

The Alloy Analyzer cannot find an instance for the this predicate, and thus returns a core consisting of lines 3 and 4. The source of the error is in line 2 where $e'.calls[i]$ refers to the post-state, which is restricted to not include i by line 3. The core does not contain the source of the error (line 2), because the Alloy language allows such joint operations resulting in an empty set. Therefore, the constraint i **in dom** $[e'.calls]$ must be added during the translation of $e'.calls[i]$, and therefore, it is easy to identify the source of the error.

Hotel. The Hotel example included in the *Software Abstractions* book about Alloy [13] did reveal a similar error in one of the post-conditions, preventing the Analyser from finding an instance. The original Alloy assertions could be used on the specification translated from VDM with only minor adjustments.

Tic Tac Toe. This example is a simple specification of the Tic tac toe game and is based on values and includes a for-statement to control the user turns. To enable analysis the complex values were converted to predicated removing the need for generators. However, for a traditional 3x3 game analysis could not complete withing 5 hours but for a board size of 2x2 analysis was possible.

Traffic. The Traffic light example was originally written as a Z specification [27] and later translated to VDM. The specification included values but they only served as test input and could thus be left out of the translation. No errors were found but the visualization of the specification made exploration of different instances easy.

6 Related Work

Various previous works have used the Alloy Analyzer to visualize or check properties of specifications expressed in different languages. However, to the author's knowledge no such attempt has been made for VDM. The translations from both B and Event-B to Alloy [15,16] both combined theorem proving with model checking and thus using Alloy to check properties. It is noted that Alloy does not have standard operations for manipulating ordinary sets that result in unnecessary long specifications: however, the current edition of Alloy includes a number of utility modules providing such features which are utilized in this work. The translation of UML with OCL to Alloy [17] identifies differences related to e.g. inheritance, collections and namespace. The latter was also encountered in our work as mentioned in in Section 3.1 under Identifiers. The translate of Z to Alloy [14] defines a semantics preserving translation for a subset of the Z language that enables automatic syntactical translation to Alloy because of the language similarities.

7 Conclusion Remarks

A key factor in applying formal methods in the software design process is automation. Automated analysis reduces errors introduced by humans and generally provides much quicker results. In this paper we have presented one approach for checking implicit VDM specifications with the Alloy Analyzer. The approach differs from earlier attempts to validate such specifications through interpretation [11]. Translating a formal language to Alloy to benefit from its analysis has been done before, but not for a higher-order model-oriented language like VDM. We have described under which circumstances a translation becomes infeasible and described how this approach has been applied to a number of specifications, and how basic errors in the specifications had remained undiscovered for years. An observation made during this work suggests that well formulated specifications at a high abstraction level tends to be easier to translate to Alloy than specifications that use unnecessary complexity or a low level of abstraction. Finally, we described a way to write assertions in the VDM notation based on expressions enabling VDM specifications to record system properties. We believe that the same principles can be adapted for other languages that share the same properties as VDM.

Acknowledgements. The author wishes to thank Daniel Jackson, Andrea Mocci, Eunsuk Kang, Peter Gorm Larsen, Nick Battle and the Software Design Group at the Computer Science and Artificial Intelligence Laboratory at Massachusetts Institute of Technology for their valuable comments and support while carrying out this work. We would also like to thank Aarhus University Forskningsfond and FondationIdella for providing the funding that made this work possible.

References

1. Bjorner, D., Jones, C.B. (eds.): The Vienna Development Method: The Meta-Language. LNCS, vol. 61. Springer, Heidelberg (1978)
2. Jones, C.B.: Systematic Software Development Using VDM, 2nd edn. Prentice-Hall International, Englewood Cliffs (1990) ISBN 0-13-880733-7
3. Fitzgerald, J.S., Larsen, P.G., Verhoef, M.: Vienna Development Method. In: Wah, B. (ed.) Wiley Encyclopedia of Computer Science and Engineering. John Wiley & Sons, Inc. (2008)
4. Larsen, P.G., Fitzgerald, J.: Recent Industrial Applications of VDM in Japan. In: Paul Boca, J.B., Larsen, P.G. (eds.) FACS 2007 Christmas Workshop: Formal Methods in Industry. Electronic Workshops in Computing, British Computer Society (December 2007)
5. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. ACM Software Engineering Notes 35(1) (January 2010)
6. Fitzgerald, J., Larsen, P.G., Sahara, S.: VDMTools: Advances in Support for Formal Modeling in VDM. ACM Sigplan Notices 43(2), 3–11 (2008)
7. Group, C.H.: The HOL System: Description (For HOL Kananaskis-4). University of Cambridge (January 2007), `http://hol.sourceforge.net/`
8. Vermolen, S.: Automatically Discharging VDM Proof Obligations using HOL. Master's thesis, Radboud University Nijmegen, Computer Science Department (August 2007)

9. Agerholm, S., Sunesen, K.: Reasoning about VDM-SL Proof Obligations in HOL. Technical report, IFAD (1999)
10. Larsen, P.G., Lassen, P.B.: An Executable Subset of Meta-IV with Loose Specification. In: Prehn, S., Toetenel, H. (eds.) VDM 1991. LNCS, vol. 552, Springer, Heidelberg (1991)
11. Fröhlich, B.: Program Generation based on Postconditions. In: Hmaza, M. (ed.) Software Enginerring, SE 1997. IASTED, ACTA Press (November 1997)
12. Fröhlich, B.: Towards Executability of Implicit Definitions. PhD thesis, TU Graz, Institute of Software Technology (September 1998)
13. Jackson, D.: Software Abstractions: Logic, Language, and Analysis, 2nd edn. MIT Press, Heyward Street (2012) ISBN-10: 0262017156
14. Malik, P., Groves, L., Lenihan, C.: Translating Z to Alloy. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 377–390. Springer, Heidelberg (2010)
15. Mikhailov, L., Butler, M.: An approach to combining B and alloy. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) B 2002 and ZB 2002. LNCS, vol. 2272, pp. 140–161. Springer, Heidelberg (2002)
16. Matos, P.J., Marques-Silva, J.: Model Checking Event-B by Encoding into Alloy. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 346–346. Springer, Heidelberg (2008)
17. Anastasakis, K., Bordbar, B., Georg, G., et al.: On challenges of Model Transformation from UML to Alloy. Software & Systems Modeling 9(1), 69–86 (2010)
18. Fitzgerald, J., Larsen, P.G.: Modelling Systems – Practical Tools and Techniques in Software Development. Cambridge University Press, The Edinburgh Building (1998) ISBN 0-521-62348-0
19. Plat, N., Larsen, P.G.: An Overview of the ISO/VDM-SL Standard. Sigplan Notices 27(8), 76–82 (1992)
20. Larsen, P.G., Pawłowski, W.: The Formal Semantics of ISO VDM-SL. Computer Standards and Interfaces 17(5-6), 585–602 (1995)
21. Jones, C., Shaw, R.: Case Studies in Systematic Software Development. Prentice Hall International (1990)
22. Aichernig, B.K.: A telephone exchange specification in VDM-SL. Technical Report IST-TEC-98-04, Technical University Graz, Austria (1998)
23. Woodcock, J., Loomes, M.: Software engineering mathematics. SEI series in software engineering. Pitman (1988) ISBN-13 9780201504248
24. Abrial, J.R.: The B Book – Assigning Programs to Meanings. Cambridge University Press (August 1996)
25. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Heyward Street (2006) ISBN-10: 0-262-10114-9
26. Fitzgerald, J., Jones, C.: Proof in the Validation of a Formal Model of a Tracking System for a Nuclear Plant. In: Bicarregui, J. (ed.) Proof in VDM: Case Studies. FACIT Series. Springer (1998)
27. Ammann, P.: A safety kernel for traffic light control. In: Haveraaen, M., Dahl, O.-J., Owe, O. (eds.) Abstract Data Types 1995 and COMPASS 1995. LNCS, vol. 1130, pp. 71–81. Springer, Heidelberg (1996)

Verification of EB^3 Specifications Using CADP

Dimitris Vekris[1,⋆], Frédéric Lang[2], Catalin Dima[1], and Radu Mateescu[2]

[1] LACL, Université Paris-Est
61, av. du Général de Gaulle, F-94010 Créteil, France
{Dimitrios.Vekris,Catalin.Dima}@u-pec.fr
[2] Inria Grenoble Rhône-Alpes and LIG – CONVECS Team
655, av. de l'Europe, Montbonnot, F-38334 Saint Ismier, France
{Frederic.Lang,Radu.Mateescu}@inria.fr

Abstract. EB^3 is a specification language for information systems. The core of the EB^3 language consists of process algebraic specifications describing the behaviour of the entities in a system, and attribute function definitions describing the entity attributes. The verification of EB^3 specifications against temporal properties is of great interest to users of EB^3. In this paper, we propose a translation from EB^3 to LOTOS NT (LNT for short), a value-passing concurrent language with classical process algebra features. Our translation ensures the one-to-one correspondence between states and transitions of the labelled transition systems corresponding to the EB^3 and LNT specifications. We automated this translation with the EB^{3}2LNT tool, thus equipping the EB^3 method with the functional verification features available in the CADP toolbox.

1 Introduction

The EB^3 method [10] is an event-based paradigm tailored for information systems (ISs). EB^3 has been used in the research projects SELKIS [19] and EB^{3}SEC [17], whose primary aim is the formal specification of ISs with security policies. In the EB^{3}SEC project, real banking industry case studies have been studied, describing interaction with brokers, customers and external financial systems. The SELKIS project deals with two case studies from the medical domain. The first one draws data records from medical imaging devices. The access to these records is done via web-based applications. The second one deals with availability and confidentiality issues for medical emergency units evolving in a great mountain range, like the Alps in that case.

A typical EB^3 specification defines entities, associations, and their respective attributes. The process algebraic nature of EB^3 enables the explicit definition of intra-entity constraints, making them easy for the IS designer to review and understand. Yet, its particular feature compared to classical process algebras,

⋆ Partially supported by SELKIS ANR Project. The LACL group is grateful to CONVECS for its warm welcome to INRIA Grenoble in 2012 and wishes to thank all its members for useful advice and discussions.

such as CSP [15], lies in the use of *attribute functions*, a special kind of recursive functions evaluated on the system execution trace. Combined with guards, attribute functions facilitate the definition of complex inter-entity constraints involving the history of events. The use of attribute functions simplifies system understanding, enhances code modularity, and streamlines maintenance. However, given that ISs are complex systems involving data management and concurrency, a rigorous design process based on formal specification using EB^3 must be completed with effective formal verification features.

Existing attempts for verifying EB^3 specifications are based on translations from EB^3 to other formal methods equipped with verification capabilities. A first line of work [13,14] focused on devising translations from EB^3 attribute functions and processes to the B method [2], which opened the way for proving invariant properties of EB^3 specifications using tools like Atelier B [6]. Another line of work concerned the verification of temporal logic properties of EB^3 specifications by means of model checking techniques. For this purpose, the formal description and verification of an IS case-study using six model checkers was undertaken in [9,5]. This study revealed the necessity of branching-time logics for accurately characterizing properties of ISs, and also the fact that process algebraic languages are suitable for describing the behaviour and synchronization of IS entities. However, no attempt of providing a systematic translation from EB^3 to a target language accepted as input by a model checker was made so far.

In this paper, we aim at filling this gap by proposing a translation from EB^3 to LNT [7], a new generation process algebraic specification language inspired from E-LOTOS [16]. As far as we know, this is the first attempt to provide a general translation from EB^3 to a classical value-passing process algebra. It is worth noticing that CSP and LNT were already considered in [9] for describing ISs, and identified as candidate target languages for translating EB^3. Since our primary objective was to provide temporal property verification features for EB^3, we focused our attention on LNT, which is one of the input languages accepted by the CADP verification toolbox [11], and hence is equipped with on-the-fly model checking for action-based, branching-time logics involving data.

At first sight, given that EB^3 has structured operational semantics based on a labelled transition system (LTS) model, its translation to a process algebra may seem straightforward. However, this exercise proved to be rather complex, the main difficulty being to translate a history-based language to a process algebra with standard LTS semantics. To overcome this difficulty, we considered alternative memory-based semantics of EB^3 [20], which were shown to be equivalent to the original trace-based semantics defined for finite-state systems in [10]. Another important ingredient of the translation was the multiway value-passing rendezvous of LNT, which enabled to obtain a one-to-one correspondence between the transitions of the two LTSs underlying the EB^3 and LNT descriptions, and hence to preserve strong bisimulation. The presence of array types and of usual programming language constructs (e.g., loops and conditionals) in LNT was also helpful for specifying the memory, the Kleene star-closure operators, and the EB^3 guarded expressions containing attribute function calls. At last,

$$
\boxed{
\begin{array}{rl}
EB^3 ::= & A_1; \ldots; A_n; S_1; \ldots; S_m \\
A ::= & f\,(\mathsf{T} : \mathcal{T}, \overline{y} : \overline{T}) : T = \textbf{match}\ \mathit{last}\,(\mathsf{T})\ \textbf{with} \\
 & \qquad \perp : v_0 \ \mathsf{|}\ \alpha_1\,(\overline{x_1}) : v_1 \ \mathsf{|}\ \ldots \ \mathsf{|}\ \alpha_q\,(\overline{x_q}) : v_q\ [\mathsf{|}\ \_ : v_{q+1}] \\
S ::= & P\,(\overline{x}) = E \\
E ::= & \lambda \mid \alpha\,(\overline{v}) \mid E_1.E_2 \mid E_1 \ \mathsf{|}\ E_2 \mid {E_0}^* \mid E_1 \ \mathsf{|[}\Delta\mathsf{]|}\ E_2 \mid \mathsf{|}x\!:\!V\!:\!E_0 \mid \\
 & \mathsf{|[}\Delta\mathsf{]|}x\!:\!V\!:\!E_0 \mid C \Rightarrow E \mid P\,(\overline{v})
\end{array}
}
$$

Fig. 1. EB^3 syntax

the constructed data types and pattern-matching mechanisms of LNT enabled a natural description of EB^3 data types and attribute functions.

We implemented our translation in the EB^{3}2LNT tool, thus making possible the analysis of EB^3 specifications using all the state-of-the-art features of the CADP toolbox, in particular the verification of data-based temporal properties expressed in MCL [18] using the on-the-fly model checker EVALUATOR 4.0.

The paper is organized as follows. Sections 2 and 3 give an overview of the EB^3 and LNT languages, respectively. Section 4 presents our translation from EB^3 to LNT, implemented by the EB^{3}2LNT translator. Section 5 shows how EB^{3}2LNT and CADP can be used for verifying the correctness requirements of an IS. Finally, Section 6 summarizes the results and draws up lines for future work.

2 The Language EB^3

The EB^3 method has been specially designed to specify the functional behaviour of ISs. A standard EB^3 specification comprises (1) a class diagram representing entity types and associations for the IS being specified, (2) a process algebra specification, denoted by *main*, describing the IS, i.e., the valid traces of execution describing its behaviour, (3) a set of attribute function definitions, which are recursive functions on the system execution trace, and (4) input/output rules to specify outputs for input traces, or SQL expressions used to specify queries on the class diagram. We limit the presentation to the process algebra and the set of attribute functions. The EB^3 syntax is presented in Figure 1 and the EB^3 trace semantics Sem_{T} [10] are given in Figure 2 as a set of rules named T_1 to T_{11}. Both figures are commented below.

Process expressions. We write $x, y, x_1, x_2, \ldots$ for variables and $v, w, v_1, v_2, \ldots$ for data expressions over user-defined domains, such as integers, Booleans and more complex domains that we do not give formally, for conciseness. Expressions are built over variables, constants, and standard operations. We also use the overlined notation as a shorthand notation for lists, e.g., $\overline{x}$ denotes a list of variables $x_1, \ldots, x_n$ of arbitrary length. An EB^3 specification consists of a set of attribute function definitions $A_1, \ldots, A_n$, and of a set of process definitions of the form "$P\,(\overline{x}) = E$", where P is a process name and E is a *process expression.*

Let *Act* be a set of *actions* written $\rho, \rho_1, \rho_2, \ldots$ and *Lab* be a set of *labels* written $\alpha, \alpha_1, \alpha_2, \ldots$ Each action ρ is either the *internal action* written λ, or a

$$(\mathrm{T}_1)\ \frac{}{\rho \xrightarrow{\rho} \surd} \qquad (\mathrm{T}_7)\ \frac{E_1 \xrightarrow{\rho} E_1' \qquad E_2 \xrightarrow{\rho} E_2'}{E_1 \mid[\Delta]\mid E_2 \xrightarrow{\rho} E_1' \mid[\Delta]\mid E_2'}\ in\,(\rho, \Delta)$$

$$(\mathrm{T}_2)\ \frac{E_1 \xrightarrow{\rho} E_1'}{E_1.E_2 \xrightarrow{\rho} E_1'.E_2} \qquad (\mathrm{T}_8)\ \frac{E_1 \xrightarrow{\rho} E_1'}{E_1 \mid[\Delta]\mid E_2 \xrightarrow{\rho} E_1' \mid[\Delta]\mid E_2}\ \neg in\,(\rho, \Delta)$$

$$(\mathrm{T}_3)\ \frac{E_2 \xrightarrow{\rho} E_2'}{\surd.E_2 \xrightarrow{\rho} E_2'} \qquad (\mathrm{T}_9)\ \frac{}{\surd \mid[\Delta]\mid \surd \xrightarrow{\lambda} \surd}$$

$$(\mathrm{T}_4)\ \frac{E_1 \xrightarrow{\rho} E_1'}{E_1 \mid E_2 \xrightarrow{\rho} E_1'} \qquad (\mathrm{T}_{10})\ \frac{E_0 \xrightarrow{\rho} E_0'}{C \Rightarrow E_0 \xrightarrow{\rho} E_0'}\ \|C\|$$

$$(\mathrm{T}_5)\ \frac{}{{E_0}^* \xrightarrow{\lambda} \surd} \qquad (\mathrm{T}_{11})\ \frac{E[\overline{x} := \overline{v}] \xrightarrow{\rho} E'}{P\,(\overline{v}) \xrightarrow{\rho} E'}\ P\,(\overline{x}) = E$$

$$(\mathrm{T}_6)\ \frac{E_0 \xrightarrow{\rho} E_0'}{{E_0}^* \xrightarrow{\rho} E_0'.{E_0}^*}$$

Fig. 2. EB^3 trace semantics Sem_{T}

visible action of the form "$\alpha\,(\overline{v})$", where $\alpha \in Lab$. An action ρ is the simplest process expression, whose semantics are given by rule T_1. The symbol $\surd$ (which is not part of the user syntax) denotes successful execution. The *trace* T (implicit in the presentation) of an EB^3 specification at a given moment consists of the sequence of visible actions executed since the start of the system. (Note therefore that λ does not appear in the trace.) At system start, the trace is empty. If T denotes the current trace and action ρ can be executed, then $\mathsf{T}.\rho$ denotes the trace just after executing ρ.

EB^3 processes can be combined with classical process algebra operators such as the *sequence* "$E_1.E_2$" ($\mathrm{T}_2, \mathrm{T}_3$), the *choice* "$E_1 \mid E_2$" ($\mathrm{T}_4$) and the *Kleene* closure "${E_0}^*$" ($\mathrm{T}_5, \mathrm{T}_6$). Rules ($\mathrm{T}_7$ to T_9) define *parallel* composition "$E_1 \mid[\Delta]\mid E_2$" of E_1, E_2 with synchronization on $\Delta \subseteq Lab$. The condition "$in\,(\rho, \Delta)$" is true iff the label of ρ belongs to Δ. The symmetric rules for choice and parallel composition have been omitted for brevity. Expressions "$E_1 \mid\mid\mid E_2$" and "$E_1 \mid\mid E_2$" are equivalent respectively to "$E_1 \mid[\emptyset]\mid E_2$" and "$E_1 \mid[Lab]\mid E_2$".

The *guarded expression* process "$C \Rightarrow E_0$" (T_{10}) can execute E_0 if the Boolean condition C holds, which is denoted by the side condition "$\|C\|$". Since C may contain calls to attribute functions, its evaluation depends on the trace obtained up to the moment when the condition is evaluated. Note that the evaluation of the guard C and the execution of the first action ρ in E_0 are simultaneous, i.e., no action is allowed in concurrent processes in the meantime. We call this property the *guard-action atomicity.* This property is essential for consistency as, by side effects, the occurrence of actions in concurrent processes could implicitly change the value of C before the guarded action has been executed.

Quantification is permitted for *choice* and *parallel* composition. If V is a set of expressions $\{v_1, \ldots, v_n\}$, "$\mid x : V : E_0$" and "$\mid[\Delta]\mid x : V : E_0$" stand respectively for "$E_0[x := v_1] \mid \ldots \mid E_0[x := v_n]$" and "$E_0[x := v_1] \mid[\Delta]\mid \ldots \mid[\Delta]\mid E_0[x := v_n]$", where "$E[x := v]$" denotes the replacement of all occurrences of x by v in E. For instance, "$\mid\mid x : \{1, 2, 3\} : a\,(x)$" stands for "$a\,(1) \mid\mid a\,(2) \mid\mid a\,(3)$". At last, named processes can be instantiated as usual (T_{11}). Given an EB^3 process expression E, we write *vars* (E) for the set of variables occurring free in E.

Attribute functions. Attribute function definitions are denoted by the symbol A in the grammar of Figure 1. Attribute functions are defined recursively on the current trace T representing the history of actions executed, with the aid of functions *last* (T) which denotes the last action of the trace, and *front* (T) which denotes the trace without its last action. The symbol $\perp$ represents the undefined value. In particular, both *last* (T) and *front* (T) match $\perp$ when the trace is empty. The symbol _ (wildcard) matches all actions not matched by any of the preceding action patterns $\alpha_1\,(\overline{x_1}), \ldots, \alpha_q\,(\overline{x_q})$. Each v_i $(i \in 0..n)$ is an expression of the same type as f's return type built over the variables $\overline{y} \cup \overline{x_i}$.

For defining formal semantics for attribute functions, the rule system of Figure 2 has to be expanded with trace and memory contexts for each process, representing the sequence of actions executed since the process was initiated, and the value of attribute functions for the current trace and any value for the rest of their arguments, stored into process memory M. Due to space limitations, we do not present the formal semantics here, but show how attribute functions are evaluated on a concrete example. The formal trace-memory semantics for attribute functions can be found in the companion paper [20].

Example. We give an example of how the trace-memory semantics work for a simplified library management system, whose specification (processes and attribute functions) in EB^3 is given in Figure 3. Process *main* is the parallel interleaving between m instances of process *book* and p instances of process *member*. Process *book* stands for a book acquisition followed by its eventual discard. The attribute function "*borrower* $(\mathsf{T},\ bId)$" looks for actions of the form "*Lend* (mId, bId)" or "*Return* (bId)" in the trace and returns the current borrower of book *bId* or $\perp$ if the book is not lent. In process book, action "*Discard* (bId)" is thus guarded to guarantee that book *bId* cannot be discarded if it is currently lent. How the use of attribute functions enhances expressiveness in the EB^3 specification of Figure 3 is discussed in [20].

We illustrate how the EB^3 specification describing the library management system is evaluated. The idea lies in the observation that attribute functions can be turned into state variables (the memory M) carrying the effect of the system trace on their corresponding values. This avoids keeping the (ever-growing) trace leading to a finite state model. If $f\,(\mathsf{T}, x_1 : T_1, \ldots, x_l : T_l)$ is an attribute function, we construct $|T_1| \times \ldots \times |T_l|$ state variables, where $|T_i|$ $(i \in 1..l)$ stands for T_i's cardinality.

```
$BID = \{b_1, \ldots, b_m\}, MID = \{m_1, \ldots, m_p\}$
$book\,(bId : BID) =$
    $Acquire\,(bId)\ .\ (borrower\,(\mathsf{T},\ bId) = \bot) \Rightarrow Discard\,(bId)$
$loan\,(mId : MID,\ bId : BID) =$
    $(borrower\,(\mathsf{T},\ bId) = \bot) \wedge (nbLoans\,(\mathsf{T}, mId) < NbLoans) \Rightarrow$
        $Lend\,(bId,\ mId)\ .\ Return\,(bId)$
$member\,(mId : MID) =$
    $Register\,(mId)\ .\ (|||\,bId : BID : loan\,(mId, bId)^*)\ .\ Unregister\,(mId)$
$main =$
    $(|||\,bId : BID : book\,(bId)^*)\ |||\ (|||\,mId : MID : member\,(mId)^*)$
```

```
$nbLoans\,(\mathsf{T} : \mathcal{T}, mId : MID) : Nat_\bot\ =$
    match $last\,(\mathsf{T})$ with
      $\bot : \bot$
    | $Lend\,(bId, mId)$ :
          $nbLoans\,(front\,(\mathsf{T}), mId) + 1$
    | $Register\,(mId) : 0$
    | $Unregister\,(mId) : \bot$
    | $Return\,(bId)$ :
        if $mId = borrower\,(\mathsf{T}, bId)$ then
          $nbLoans\,(front\,(\mathsf{T}), mId) - 1$
        else $nbLoans\,(front\,(\mathsf{T}), mId)$ end if
    | _: $nbLoans\,(front\,(\mathsf{T}), mId)$
    end match
```

```
$borrower\,(\mathsf{T} : \mathcal{T}, bId : BID) : MID_\bot\ =$
    match $last\,(\mathsf{T})$ with
      $\bot : \bot$
    | $Lend\,(bId, mId) : mId$
    | $Return\,(bId) : \bot$
    | _: $borrower\,(front\,(\mathsf{T}), bId)$
    end match
```

Fig. 3. EB^3 specification of a library management system

As an example, we set $m = p = NbLoans = 2$, i.e. we consider two books b_1 and b_2, and two members m_1 and m_2. The memory has four cells: $\mathsf{M} = (borrower\,[b_1], borrower\,[b_2], nbLoans\,[m_1], nbLoans\,[m_2])$. The first two cells keep the two values of the attribute function $borrower\,(\mathsf{T}, \bullet)$ for a given trace T, and the last two keep the values of $nbLoans\,(\mathsf{T}, \bullet)$. After every step, the new value of each cell can be calculated from the previous memory and the action that has just been executed. The memory is initially set to $(\bot, \bot, \bot, \bot)$. After the trace "$Acquire\,(b_1).Acquire\,(b_2).Register\,(m_1).Register\,(m_2)$" the memory contains $(\bot, \bot, 0, 0)$. If action "$Lend\,(b_1, m_1)$" is then executed, the new memory is $(m_1, \bot, 1, 0)$. For instance, the new value m_1 for $borrower\,[b_1]$ is obtained from the rule "$Lend\,(bId, mId) : mId$" in the definition of the attribute function *borrower* (see Fig. 3), and the new value 1 for $nbLoans\,[m_1]$ by the rule "$Lend\,(bId, mId) : nbLoans\,(front\,(\mathsf{T}), mId) + 1$" of the attribute function *nbLoans*, where the value of $nbLoans\,(front\,(\mathsf{T}), m_1)$ corresponds to the value of $nbLoans\,[m_1]$ in the previous memory state (value 0).

3 The Language LNT

LNT aims at providing the best features of imperative and functional programming languages and value-passing process algebras. It has a user friendly syntax and formal operational semantics defined in terms of labeled transition systems (LTSs). LNT is supported by the LNT.OPEN tool of CADP, which allows the on-the-fly exploration of the LTS corresponding to an LNT specification.

We present the fragment of LNT that serves as the target of our translation. Its syntax is given in Figure 4. LNT terms denoted by B are built from actions, choice (**select**), conditional (**if**), sequential composition (**;**), breakable loop (**loop** and **break**) and parallel composition (**par**). Communication is carried out by rendezvous on gates, written G, G_1, ..., G_n, and may be guarded using Boolean conditions on the received values (**where** clause). LNT allows multiway rendezvous with bidirectional (send/receive) value exchange on the same gate occurrence, each offer O being either a send offer (!) or a receive offer (?), independently of the other offers. Expressions E are built from variables, type constructors, function applications and constants. Labels L identify loops, which can be escaped using "**break** L" from inside the loop body. Processes are parameterized by gates and data variables. LNT semantics are formally defined in SOS style in [7].

$$\boxed{\begin{array}{lll}
B & ::= & \textbf{stop} \mid \textbf{null} \mid G\,(O_1,\ldots,O_n)\ \textbf{where}\ E \mid B_1; B_2 \\
 & \mid & \textbf{if}\ E\ \textbf{then}\ B_1\ \textbf{else}\ B_2\ \textbf{end if} \mid \textbf{var}\ x\!:\!T\ \textbf{in}\ B\ \textbf{end var} \mid x := E \mid \\
 & \mid & \textbf{loop}\ L\ \textbf{in}\ B\ \textbf{end loop} \mid \textbf{break}\ L \mid \textbf{select}\ B_1\ \texttt{[]}\ \ldots\ \texttt{[]}\ B_n\ \textbf{end select} \\
 & \mid & \textbf{par}\ G_1,\ldots,G_n\ \textbf{in}\ B_1\ \texttt{||}\ \ldots\ \texttt{||}\ B_n\ \textbf{end par} \mid P\,\texttt{[}G_1,\ldots,G_n\texttt{]}\ (E_1,\ldots,E_n) \\
O & ::= & \texttt{!}E \mid \texttt{?}x
\end{array}}$$

Fig. 4. LNT syntax (limited to the fragment used in this paper)

4 Translation from EB^3 to LNT

Principles. Our translation of EB^3 relies on the trace-memory semantics. Thus, we explicitly model in LNT a memory, which stores the state variables corresponding to attribute functions (we call these variables *attribute variables*) and is modified each time an action is executed.

Assuming n attribute functions $f_1, \ldots, f_n$, we model the memory as a process M placed in parallel with the rest of the system (a common approach for modeling global variables in process algebras), which manages for each attribute function f_i an attribute variable (also named f_i) that encodes the function. To read the values of these attribute variables (i.e., to evaluate the attribute functions), processes need to communicate with the memory M, and every action must have an immediate effect on the memory (so as to reflect the immediate effect on the execution trace). To achieve this, the memory process synchronizes with the rest of the system on every possible action of the system (including λ, to which we associate an LNT gate also written λ in abstract syntax for convenience), and updates its attribute variables accordingly. The list of attribute variables $\overline{f} = (f_1, \ldots, f_n)$ is added as a supplementary offer on each EB^3 action $\alpha\,(\overline{v})$, so that attribute variables can be directly accessed to evaluate the guard associated to the action, wherever needed, while guaranteeing the guard-action atomicity. Therefore, every action $\alpha\,(\overline{v})$ will be encoded in LNT as $\alpha\,(\texttt{!}\overline{v}, \texttt{?}\overline{f})$, and synchronized with an action of the form $\alpha\,(\texttt{?}\overline{x}, \texttt{!}\overline{f})$ in the memory process M, thus taking benefit of bidirectional value exchange during the rendezvous.

Translation of attribute functions. To formalize the translation, we assume $Lab = \{\alpha_1, \ldots, \alpha_q\}$ (not including λ), each α_j has formal parameters $\overline{x_j}$, $\{f_1, \ldots, f_n\}$ is the set of attribute functions, and each f_i is uniquely defined by the set of formal parameters $\overline{y_i}$ and the set of data expressions $w_i^0, \ldots, w_i^q$, such that:

$$f_i\,(\mathsf{T}, \overline{y_i}) = \textbf{match}\ \mathit{last}\,(\mathsf{T})\ \textbf{with}\ \bot : w_i^0 \mid \alpha_1\,(\overline{x_1}) : w_i^1 \mid \ldots \mid \alpha_q\,(\overline{x_q}) : w_i^q$$

We also assume that the attribute functions are ordered, so that for all $h \in 1..n, i \in 1..n, j \in 1..q$, every function call of the form $f_h\,(\mathsf{T}, \ldots)$ occurring in w_i^j satisfies $h < i$ and every call of the form $f_h\,(\mathit{front}\,(\mathsf{T}), \ldots)$ satisfies $h \geq i$. Such an ordering can be constructed if the EB^3 specification does not contain circular dependencies between function calls, which would potentially lead to infinite attribute function evaluation. In particular, the definition of an attribute function f_i cannot contain recursive calls of the form "$f_i\,(\mathsf{T}, \ldots)$", but only recursive calls of the form "$f_i\,(\mathit{front}\,(\mathsf{T}), \ldots)$". Note that this does not limit the expressiveness of EB^3 attribute functions, because every recursive computation on data expressions only (which keeps the trace unchanged) can be described using standard functions and not attribute functions.

Ordering attribute functions in this way allows the memory to be updated consistently, from f_1 to f_n in turn. At every instant, already-updated values correspond to calls of the form $f_h\,(\mathsf{T}, \ldots)$ (the value of f_h on the current trace), whereas calls of the form $f_h\,(\mathit{front}\,(\mathsf{T}), \ldots)$ are replaced by accesses to a copy $\overline{f'}$ of the memory $\overline{f}$, which was made before starting the update. This encoding thus enables the trace parameter to be discharged from function calls, ensuring that while updating f_i, accesses to f_h with $h < i$ necessarily correspond to calls with parameter T.

Process M is defined in Figure 5. It runs an infinite loop, which "listens" to all possible actions α_j of the system. Each attribute variable f_i is an array with l_i dimensions, where l_i is the arity of the attribute function f_i minus 1 (because the trace parameter is now discharged). Each dimension of the array f_i thus corresponds to one formal parameter in $\overline{y_i}$, so that $f_i\,\texttt{[}\textbf{ord}\,(v_1)\texttt{]} \ldots \texttt{[}\textbf{ord}\,(v_{l_i})\texttt{]}$ encodes the current value of $f_i\,(\mathsf{T}, v_1, \ldots, v_{l_i})$, where $\textbf{ord}\,(v)$ is a predefined LNT function that denotes the *ordinate* of value v, i.e., a unique number between 1 and the cardinal of v's type. For each type T we assume the existence of functions first_T that returns the first element of type T, last_T that returns the last element of type T, and $\mathit{next}_T\,(x)$ that returns the successor of x in type T (following the total order induced by $\textbf{ord}$). Such functions are available in LNT

```
process $M$ [$\alpha_1, \ldots, \alpha_q, \lambda$ : any] is
  var $\overline{f}, \overline{f'}$ : type ($\overline{f}$),
      $\overline{y_1}$ : type ($\overline{y_1}$), ..., $\overline{y_n}$ : type ($\overline{y_n}$), $\overline{x_1}$ : type ($\overline{x_1}$), ..., $\overline{x_q}$ : type ($\overline{x_q}$) in
    $upd_1^0$; ... ; $upd_n^0$;
    loop
      $\overline{f'} := \overline{f}$ (* $f_i'$[$ord(\overline{v})$] will encode $f_i(front(\mathsf{T}), \overline{v})$ during memory update *)
      select
      $\alpha_1$ (?$\overline{x_1}$, !$\overline{f}$); $upd_1^1$; ... ; $upd_n^1$
      [] ... []
      $\alpha_q$ (?$\overline{x_q}$, !$\overline{f}$); $upd_1^q$; ... ; $upd_n^q$
      [] $\lambda$ (!$\overline{f}$)
      end select
    end loop
  end var
end process
$upd_i^j \doteq enum(\overline{y_i}, f_i$[$ord(\overline{y_i})$] $:= mod(w_i^j))$
$enum([\,], B) \doteq B$
$enum(x :: \overline{y}, B) \doteq x := first_T$;
                    loop $L_x$ in
                      $enum(\overline{y}, B)$
                      if $x \neq last_T$ then $x := next_T(x)$ else break $L_x$ end if
                    end loop where $T = \text{type}(x)$
$v$[$ord(\overline{y})$] $\doteq v$[ord $(y_1)$] ... [ord $(y_l)$], ?$\overline{y}$ = (?$y_1, \ldots,$ ?$y_l$), where $\overline{y} = (y_1, \ldots, y_l)$
$mod(E) \doteq E$ [ $f_i(\mathsf{T}, \overline{v_i}) := f_i$[$ord(\overline{v_i})$], $f_i(front(\mathsf{T}), \overline{v_i}) := f_i'$[$ord(\overline{v_i})$] | $i \in 1..n$ ]
```

Fig. 5. LNT code for the memory process implementing attribute functions

for all finite types. Function *mod* transforms an expression E by syntactically replacing function calls by array accesses, while discharging the trace parameter as explained above.

Upon synchronisation on action $\alpha_j\,(?\overline{x_j}, !\overline{f})$ with the LNT process corresponding to EB^3's *main* process (see translation of processes below), the values of all attribute variables f_i $(i \in 1..n)$ are updated using function upd_i^j.

Translation of processes. We define a translation function t from an EB^3 process expression to an LNT process. Most EB^3 constructs are process algebra constructs with a direct correspondence in LNT. The main difficulty arises in the translation of guarded process expressions of the form "$C \Rightarrow E_0$" in a way that guarantees the guard-action atomicity. This led us to consider a second parameter for the translation function t, namely the condition C, whose evaluation is delayed until the first action occurring in the process expression E_0. The definition of $t\,(E, C)$ is given in Figure 6. An EB^3 specification E_0 will then be translated into "**par** $\alpha_1, \ldots, \alpha_q, \lambda$ **in** $t\,(E_0, \text{true})$ || M [$\alpha_1, \ldots, \alpha_q, \lambda$] **end par**" and every process definition of the form "$P\,(\overline{x}) = E$" will be translated into the process "**process** P [$\alpha_1, \ldots, \alpha_q, \lambda$: **any**] ($\overline{x}$: type ($\overline{x}$)) **is** $t\,(E, \text{true})$ **end process**", where $\{\alpha_1, \ldots, \alpha_q\} = Lab$. The rules of Figure 6 can be commented as follows:

$$\begin{aligned}
t(\lambda, C) &= \lambda\,(?\overline{f})\ \textbf{where}\ \mathit{mod}\,(C) && (1)\\
t(\alpha\,(\overline{v}), C) &= \alpha\,(\overline{v}, ?\overline{f})\ \textbf{where}\ \mathit{mod}\,(C) && (2)\\
t(E_1.E_2, C) &= t(E_1, C);\ t(E_2, \text{true}) && (3)\\
t(C' \Rightarrow E_0, C) &= t(E_0, C\ \textbf{andthen}\ C') && (4)\\
t(E_1 \mid E_2, C) &= \textbf{select}\ t(E_1, C)\ \texttt{[]}\ t(E_2, C)\ \textbf{end select} && (5)\\
t(\mid x\colon V \colon E_0, C) &= \textbf{var}\ x := \textbf{any}\ V;\ t(E_0, C)\ \textbf{end var} && (6)\\
t({E_0}^*, \text{true}) &= \textbf{loop}\ L_{E_0}\ \textbf{in}\\
&\qquad \textbf{select}\\
&\qquad\quad \lambda\,(?\overline{f});\ \textbf{break}\ L_{E_0}\ \texttt{[]}\ t(E_0, \text{true})\\
&\qquad \textbf{end select}\\
&\quad \textbf{end loop} && (7)\\
t(E_1 \mid[\Delta]\mid E_2, \text{true}) &= \textbf{par}\ \Delta\ \textbf{in}\ t(E_1, \text{true})\ \mid\mid\ t(E_2, \text{true})\ \textbf{end par} && (8)\\
t(\mid[\Delta]\mid x \colon V \colon E_0, \text{true}) &= \textbf{par}\ \Delta\ \textbf{in}\ E_0[x := v_1]\ \mid\mid\ \ldots\ \mid\mid\ E_0[x := v_n]\ \textbf{end par}\\
&\qquad \text{where } V = \{v_1, \ldots, v_n\} && (9)\\
t(P\,(\overline{v}), \text{true}) &= P\ [\alpha_1, \ldots, \alpha_q, \lambda]\ (\overline{v}) && (10)
\end{aligned}$$

In all other cases:

$$t(E_0, C) = \begin{cases} \textbf{if}\ \mathit{mod}\,(C)\ \textbf{then}\ t(E_0, \text{true})\ \textbf{else stop end if} \\ \quad \text{if } C \text{ does not use attribute functions} \\ \textbf{par}\ \alpha_1, \ldots, \alpha_q, \lambda\ \textbf{in} \\ \quad t(E_0, \text{true}) \\ \mid\mid\ \mathit{pr}_C\ [\alpha_1, \ldots, \alpha_q, \lambda]\ (\mathit{vars}\,(C)) \\ \textbf{end par} \qquad \text{otherwise} \end{cases} \qquad (11)$$

Fig. 6. Translation from EB^3 process to LNT process

- Rule (1) translates the λ action. Note that λ cannot be translated to the empty LNT statement **null**, because execution of λ may depend on a guard C, whose evaluation requires the memory to be read, so as to get attribute variable values. This is done by the LNT communication action $\lambda\,(?\overline{f})$. The guard C is evaluated after replacing calls to attribute functions (all of which have the form $f_i\,(\mathsf{T}, \overline{v_i})$) by the appropriate attribute variables, using function *mod* defined in Figure 5. Rule (2) is similar but handles visible actions.
- Rule (3) translates EB^3 sequential composition into LNT sequential composition, passing the evaluation of C to the first process expression.
- Rule (4) makes a conjunction between the guard of the current process expression with the guard already accumulated from the context.
- Rules (5) and (6) translate the choice and quantified choice operators of EB^3 into their direct LNT counterpart.
- Rule (7) translates the Kleene closure into a combination of LNT loop and select, following the identity ${E_0}^* = \lambda \mid E_0.{E_0}^*$.
- Rule (8) translates EB^3 parallel composition into LNT parallel composition.

- Rule (9) translates EB^3 quantified parallel composition into LNT parallel composition by expanding the type V of the quantification variable, since LNT does not have a quantified parallel composition operator.
- Rule (10) translates an EB^3 process call into the corresponding LNT process call, which requires gates to be passed as parameters.
- Rules (7) to (10) only apply when the guard C is trivially true. In the other cases, we must apply rule (11), which generates code implementing the guard. If C does not use attribute functions, i.e., does not depend on the trace, then it can be evaluated immediately without communicating with the memory process (first case). Otherwise, the guard evaluation must be delayed until the first action of the process expression E_0. When E_0 is either a Kleene closure, a parallel composition, or a process call, identifying its first action syntactically is not obvious. One solution would consist in expanding E_0 into a choice in which every branch has a fixed initial action[1], to which the guard would be added. We preferred an alternative solution that avoids the potential combinatorial explosion of code due to static expansion. A process pr_C (defined in Fig. 7) is placed in parallel to $t\,(E_0, \text{true})$ and both processes synchronize on all actions. Process pr_C imposes on $t\,(E_0, \text{true})$ the constraint that the first executed action must satisfy the condition C (**then** branch). For subsequent actions, the condition is relaxed (**else** branch).

The following example illustrates and justifies the use of process pr_C as a means to solve the guard-action atomicity problem. Consider the EB^3 system "$C \Rightarrow \mathit{Lend}\,(b_1,\ m_1)\ |||\ \mathit{Return}\,(b_2)$", where C denotes the Boolean condition "$\mathit{borrower}\,(\mathsf{T},\ b_1) = \bot \wedge \mathit{nbLoans}\,(\mathsf{T},\ m_1) < \mathit{NbLoans}$" and $\mathit{Lab} = \{\mathit{Lend}, \mathit{Return}\}$. The LNT code corresponding to this system is the following:

```
par $Lend, Return, \lambda$ in
   par $Lend, Return, \lambda$ in
      par $Lend\,(b_1,\ m_1, ?\overline{f})$ || $Return\,(b_2, ?\overline{f})$ end par
   || $pr_C$ [$Lend, Return, \lambda$] $(b_1,\ m_1)$
   end par
|| $M$ [$Lend, Return, \lambda$]
end par
```

The first action executed by this system may be either *Lend* or *Return*. We consider the case where *Lend* is executed first. According to the LNT semantics, it results from the multiway synchronization of the following three actions:

- "$\mathit{Lend}\,(b_1,\ m_1, ?\overline{f})$" in the above expression,
- "$\mathit{Lend}\,(?b, ?m, ?\overline{f})$ **where** $\mathit{borrower}[\mathbf{ord}(b_1)] = \bot \wedge \mathit{nbLoans}[\mathbf{ord}(m_1)] < \mathit{NbLoans}$" in process pr_C (at this moment, *start* is true, see Fig. 7), and
- "$\mathit{Lend}\,(?b, ?m, !\overline{f})$" in process M (see Fig. 5).

Thus, in pr_C at synchronization time, $\overline{f}$ is an up-to-date copy of the memory stored in M, $b = b_1$, and $m = m_1$. The only condition for the synchronization to

[1] Such a form, commonly called *head normal form* [3], is used principally in the context of the process algebra ACP [4] to analyse the behaviour of recursive processes.

```
process $pr_C$ [$\alpha_1, \ldots, \alpha_q, \lambda$ : any] ($vars(C)$ : $type(vars(C))$) is
var $start$ : bool, $\overline{x_1}$ : type($\overline{x_1}$), ..., $\overline{x_q}$ : type($\overline{x_q}$) in
   $start$ := true;
   loop $L$ in select
      if $start$ then
         $start$ := false;
         select
            $\alpha_1$ (?$\overline{x_1}$, ?$\overline{f}$) where $mod(C)$
            [] ... []
            $\alpha_q$ (?$\overline{x_q}$, ?$\overline{f}$) where $mod(C)$
            []
            $\lambda$ (?$\overline{f}$) where $mod(C)$
         end select
      else
         select
            $\alpha_1$ (?$\overline{x_1}$, ?$\overline{f}$)
            [] ... []
            $\alpha_q$ (?$\overline{x_q}$, ?$\overline{f}$)
            []
            $\lambda$ (?$\overline{f}$)
         end select
      end if
   [] break $L$ end select end loop
end var
end process
```

Fig. 7. Process pr_C

occur is the guard $mod(C)$, whose value is computed using the up-to-date copy $\overline{f}$ of the memory. In case $mod(C)$ evaluates to true, no other action (susceptible to modifying $\overline{f}$) can occur between the evaluation of $mod(C)$ and the occurrence of *Lend* as both happen synchronously, thus achieving the guard-action atomicity. Once *Lend* has occurred, *Return* can occur without any condition, as the value of *start* has now become false.

Theorem 1. *Let E, E' be* EB^3 *process expressions,* T *be the current trace,* $\overline{f}$ *be the set of attribute functions, and* $\rho \in Act$. *Then* $E \xrightarrow{\rho(\overline{x})} E'$ *if and only if:*

$$t(E, true) \xrightarrow{\rho(\overline{x}, \overline{f})} t(E', true) \wedge (\forall f_i \in \overline{f})\ (\forall \overline{v})\ f_i(\mathsf{T}, \overline{v}) = f_i\,[ord(\overline{v})]\,.$$

The proof strategy for Theorem 1 relies on the existence of a bisimulation between each EB^3 specification and its corresponding LNT translation. It works by providing a match between the reduction rules of EB^3 [20] and the corresponding LNT rules [7].

We developed an automatic translator tool from EB^3 specifications to LNT, named EB^{3}2LNT, implemented using the Ocaml Lex/Yacc compiler construction technology. It consists of about 900 lines of OCaml code. We applied EB^{3}2LNT on a benchmark of EB^3 specifications, which includes variations of the library management system (examined in its simplest version in Section 2) and a bank account management system.

We noticed that, for each EB^3 specification, the code size of the equivalent LNT specification is twice as big. Part of this expansion is caused by the fact that LNT is more structured than EB^3: LNT requires more keywords and gates have to be declared and passed as parameters to each process call. By looking at the rules of Figure 6, we can see that the other causes of expansion are rule (5), which duplicates the condition C, and rule (9), which duplicates the body E_0 of the quantified parallel composition operator "$|[\Delta]|\, x : V : E_0$" as many times as there are elements in the set V. Both expansions are linear in the size of the source EB^3 code. However, in the case of a nested parallel composition "$|[\Delta_1]|\, x_1 : V_1 : \ldots |[\Delta_n]|\, x_n : V_n : E_0$", the expansion factor is as high as the product of the number of elements in the respective sets $V_1, \ldots, V_n$, which may be large. If E_0 is a big process expression, the expansion can be limited by encapsulating E_0 in a parameterized process "$P_{E_0}\,(x_1, \ldots, x_n)$" and replacing duplicated occurrences of E_0 by appropriate instances of P_{E_0}.

5 Case Study

We illustrate below the application of the EB^{3}2LNT translator in conjunction with CADP for analyzing an extended version of the IS library management system, whose description in EB^3 can be found in Annex C of [12]. With respect to the simplified version presented in Section 2, the IS enables e.g., members to renew their loans and to reserve books, and their reservations to be cancelled or transferred to other members on demand. The desired behaviour of this IS was characterized in [9] as a set of 15 requirements expressed informally as follows:

R1. A book can always be acquired by the library when it is not currently acquired.
R2. A book cannot be acquired by the library if it is already acquired.
R3. An acquired book can be discarded only if it is neither borrowed nor reserved.
R4. A person must be a member of the library in order to borrow a book.
R5. A book can be reserved only if it has been borrowed or already reserved by some member.
R6. A book cannot be reserved by the member who is borrowing it.
R7. A book cannot be reserved by a member who is reserving it.
R8. A book cannot be lent to a member if it is reserved.
R9. A member cannot renew a loan or give the book to another member if the book is reserved.
R10. A member is allowed to take a reserved book only if he owns the oldest reservation.
R11. A book can be taken only if it is not borrowed.
R12. A member who has reserved a book can cancel the reservation at anytime before he takes it.
R13. A member can relinquish library membership only when all his loans have been returned and all his reservations have either been used or cancelled.
R14. Ultimately, there is always a procedure that enables a member to leave the library.

R15. A member cannot borrow more than the loan limit defined at the system level for all users.

We expressed all the above requirements using the property specification language MCL [18]. MCL is an extension of the alternation-free modal μ-calculus [8] with action predicates enabling value extraction, modalities containing extended regular expressions on transition sequences, quantified variables and parameterized fixed point operators, programming language constructs, and fairness operators encoding generalized Büchi automata. These features make possible a concise and intuitive description of safety, liveness, and fairness properties involving data, without sacrificing the efficiency of on-the-fly model checking, which has a linear-time complexity for the dataless MCL formulas [18].

We show below the MCL formulation of two requirements from the list above, which denote typical safety and liveness properties. Requirement R2 is expressed in MCL as follows:

$$[\,\mathbf{true}^*.\{\texttt{ACQUIRE ?}B : \mathbf{string}\}.(\mathbf{not}\ \{\texttt{DISCARD !}B\})^*.\{\texttt{ACQUIRE !}B\}\,]\ \mathbf{false}$$

This formula uses the standard *safety* pattern "$[\beta]$ **false**", which forbids the existence of transition sequences matching the regular formula β. Here the undesirable sequences are those containing two *Acquire* operations for the same book B without a *Discard* operation for B in the meantime. The regular formula $\mathbf{true}^*$ matches a subsequence of (zero or more) transitions labeled by arbitrary actions. Note the use of the construct "?B : **string**", which matches any string and extracts its value in the variable B used later in the formula. Therefore, the above formula captures all occurrences of books carried by *Acquire* operations in the model. Requirement R12 is formulated in MCL as follows:

$$\begin{array}{l}[\,\mathbf{true}^*.\{\texttt{RESERVE ?}M : \mathbf{string}\ \texttt{?}B : \mathbf{string}\}.\\ \ (\mathbf{not}\ (\{\texttt{TAKE !}M\ \texttt{!}B\}\ \mathbf{or}\ \{\texttt{TRANSFER !}M\ \texttt{!}B\}))^*\,]\\ \quad\langle\,(\mathbf{not}\ (\{\texttt{TAKE !}M\ \texttt{!}B\}\ \mathbf{or}\ \{\texttt{TRANSFER !}M\ \texttt{!}B\}))^*.\ \{\texttt{CANCEL !}M\ \texttt{!}B\}\,\rangle\ \mathbf{true}\end{array}$$

This formula denotes a *liveness* property of the form "$[\beta_1]\ \langle\beta_2\rangle$ **true**", which states that every transition sequence matching the regular formula β_1 (in this case, book B has been reserved by member M and subsequently neither taken nor transferred) ends in a state from which there exists a transition sequence matching the regular formula β_2 (in this case, the reservation can be cancelled before being taken or transferred).

Using EB^{3}2LNT, we translated the EB^3 specification of the library management system to LNT. The resulting specification was checked against all the 15 requirements, formulated in MCL, using the EVALUATOR 4.0 model checker of CADP. The experiments were performed on an Intel(R) Core(TM) i7 CPU 880 at 3.07GHz. Table 1 shows the results for several configurations of the IS, obtained by instantiating the number of books (m) and members (p) in the IS. All requirements were shown to be valid on the IS specification. The second and third line of the table indicate the number of states and transitions of the LTS corresponding to the LNT specification. The fourth line gives the time needed to

Table 1. Model checking results for the library management system

(m, p)	(3,2)	(3,3)	(3,4)	(4,3)
states	1,002	182,266	8,269,754	27,204,016
trans.	5,732	1,782,348	105,481,364	330,988,232
time	1.9s	14.4s	31’39s	140’22s
R1	0.3s	1.8s	5’19s	20’13s
R2	0.2s	2.9s	9’26s	36’7s
R3	0.2s	9.4s	97’46s	26’47s
R4	0.2s	1.7s	5’15s	18’40s
R5	0.2s	2.2s	6’46s	21’52s
R6	0.2s	4.1s	38’30s	10’19s
R7	0.2s	7.4s	65’22s	24’33s
R8	0.2s	2.2s	6’52s	22’27s
R9	0.2s	2.3s	6’38s	22’29s
R10	0.3s	13.3s	43’59s	62’07s
R11	0.3s	2.5s	6’36s	22’14s
R12	0.3s	4.0s	10’47s	45’09s
R13	0.4s	4.3s	11’46s	1’07s
R14	0.3s	3.6s	10’41s	37’33s
R15	0.2s	2.8s	7’53s	28’56s

generate the LTS and the other lines give the verification time for each requirement. Note that the number of states generated increases with the size of m and p as EVALUATOR 4.0 applies explicit techniques for state space generation.

6 Conclusion

We proposed an approach for equipping the EB^3 method with formal verification capabilities by reusing already available model checking technology. Our approach relies upon a new translation from EB^3 to LNT, which provides a direct connection to all the state-of-the-art verification features of the CADP toolbox. The translation, based on alternative memory semantics of EB^3 [20] instead of the original trace semantics [10], was automated by the EB^{3}2LNT translator and validated on several examples of typical ISs. So far, we experimented only the model checking of MCL data-based temporal properties on EB^3 specifications. However, CADP also provides extensive support for equivalence checking and compositional LTS construction, which can be of interest to IS designers.

As future work, we plan to provide a formal proof of the translation from EB^3 to LNT, which could serve as reference for translating EB^3 to other process algebras as well. We also plan to study abstraction techniques for verifying properties regardless of the number of entity instances that participate in the IS, following the approaches for parameterized model checking [1]. In particular, we will observe how the insertion of new functionalities into an IS affects this issue, and we will formalize this in the context of EB^3 specifications.

References

1. Abdulla, P.A., Bouajjani, A., Jonsson, B., Nilsson, M.: Handling Global Conditions in Parameterized System Verification. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 134–145. Springer, Heidelberg (1999)
2. Abrial, J.-R.: The B-Book - Assigning programs to meanings. Cambridge University Press (2005)
3. Bergstra, J.A., Ponse, A., Smolka, S.A.: Handbook of Process Algebra. Elsevier (2001)
4. Bergstra, J.A., Klop, J.W.: Algebra of Communicating Processes with Abstraction. TCS 37, 77–121 (1985)
5. Chossart, R.: Évaluation d'outils de vérification pour les spécifications de systèmes d'information. Master's thesis, Université de Sherbrooke (2010)
6. ClearSy. Atelier B, `http://www.atelierb.societe.com`
7. Champelovier, D., Clerc, X., Garavel, H., Guerte, Y., McKinty, C., Powazny, V., Lang, F., Serwe, W., Smeding, G.: Reference Manual of the LOTOS NT to LOTOS Translator - Version 5.4. In: INRIA/VASY (2011)
8. Allen Emerson, E., Lei, C.-L.: Efficient Model Checking in Fragments of the Propositional Mu-Calculus. In: Proc. of LICS, pp. 267–278 (1986)
9. Frappier, M., Fraikin, B., Chossart, R., Chane-Yack-Fa, R., Ouenzar, M.: Comparison of model checking tools for information systems. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 581–596. Springer, Heidelberg (2010)
10. Frappier, M., St.-Denis, R.: EB^3: an entity-based black-box specification method for information systems. Software and System Modeling 2(2), 134–149 (2003)
11. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2010: A toolbox for the construction and analysis of distributed processes. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 372–387. Springer, Heidelberg (2011)
12. Gervais, F.: Combinaison de spécifications formelles pour la modélisation des systèmes d'information. PhD thesis, Université de Sherbrooke (2006)
13. Gervais, F., Frappier, M., Laleau, R.: Synthesizing B Specifications from EB^3 Attribute Definitions. In: Romijn, J.M.T., Smith, G.P., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 207–226. Springer, Heidelberg (2005)
14. Gervais, F., Frappier, M., Laleau, R.: Refinement of EB^3 Process Patterns into B Specifications. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 201–215. Springer, Heidelberg (2006)
15. Hoare, C.A.R.: Communicating Sequential Processes. Commun. ACM 21(8), 666–677 (1978)
16. ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard number 15437:2001, International Organization for Standardization — Information Technology, Genève (2001)
17. Jiague, M.E., Frappier, M., Gervais, F., Konopacki, P., Laleau, R., Milhau, J., St-Denis, R.: Model-Driven Engineering of Functional Security Policies. In: Proc. of ICEIS, pp. 374–379 (2010)
18. Mateescu, R., Thivolle, D.: A model checking language for concurrent value-passing systems. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 148–164. Springer, Heidelberg (2008)
19. Milhau, J., Idani, A., Laleau, R., Labiadh, M.A., Ledru, Y., Frappier, M.: Combining UML, ASTD and B for the formal specification of an access control filter. Journal of Innovations in Systems and Software Engineering 7, 303–313 (2011)
20. Vekris, D., Dima, C.: Efficient Operational Semantics for EB^3 for Verification of Temporal Properties. In: Proc. of FSEN. Springer (to appear, 2013)

Knowledge for the Distributed Implementation of Constrained Systems

(Extended Abstract)

Susanne Graf[1] and Sophie Quinton[2]

[1] Université Joseph Fourier, VERIMAG
[2] Institute of Computer and Network Engineering, TU Braunschweig

Abstract. Deriving distributed implementations from global specifications has been extensively studied for different application domains, under different assumptions and constraints. We explore here the knowledge perspective: a process decides to take a local action when it has the *knowledge* to do so. We discuss typical knowledge atoms that are useful for expressing local enabling conditions with respect to different notions of correctness, as well as different means for obtaining knowledge and for representing it locally in an efficient manner. Our goal is to use such a knowledge-based representation of the distribution problem for either deriving distributed implementations automatically from global specifications on which some constraint is enforced, or for improving the efficiency of existing protocols by exploiting local knowledge. We also argue that such a knowledge-based presentation helps achieving the necessary correctness proofs.

1 Introduction

Building correct distributed systems is a challenging issue where the complexity of global verification is bound to be unmanageable. An interesting solution to this consists in starting from a centralized specification of the system under construction, verifying all properties of interest on this centralized specification — which has a much lower complexity than the verification on a distributed implementation — and finally derive a distributed implementation using some correct-by-construction approach. Note that this topic is related to distributed control, where the objective is to enforce in a distributed manner some global constraint on a plant. Deriving such a distributed controller directly is difficult, and the correctness of the resulting controller is difficult to prove. A more feasible approach in this context is to first construct a global controller, which then is transformed into distributed one, again using some correct-by-construction approach. In this paper, we consider a similar design methodology:

1. We suppose given a global (centralized) specification S of the system to be implemented and a global constraint Ψ that has to be enforced. Our first issue is to construct a global controller that enforces a controlled specification S^{Ψ}.
2. As a second step, we may perform analysis on our global controlled specification S^{Ψ} and make sure that it satisfies the required safety properties.

3. Finally, we have to execute S^{Ψ} in a distributed way. That is, we must decompose the specification into k independent processes $S_1, \ldots, S_k$ executing on a distributed platform, either totally agnostic of each other or communicating — in a limited way — through the communication system provided by this platform. The system obtained as the composition of the local specifications $S_1, \ldots, S_k$ is denoted S_{dis}. This distributed implementation must behave according to the global controlled specification S^{Ψ}. Note that here, the notion of correctness defining what it means to *behave according to* a specification depends on the type of properties that we verify in the second phase.

We use the concept of *knowledge* [7] to express how a process can decide which of its feasible transitions it should execute (if any) to satisfy the correctness criterion. In a series of recent papers [2,9,3,10,15,4] it has been proposed to construct such knowledge by means of a global analysis in order to distribute the controller enforcing constraint Ψ, while relying for the distribution of the specification on some standard protocol such as α-core [17]. In particular, global constraints defined by a priority order amongst global transitions were considered. On the other hand, [11] proposed a protocol that is similar to α-core but can handle global priorities directly, yet not exploiting knowledge explicitly. This allows obtaining a distributed implementation of priority systems without a prior global analysis. We argue here that the two approaches may be conveniently combined by:

– applying a global static analysis in order to compute knowledge that is useful either for the distributed implementation of S or for achieving the control constraint Ψ (respectively directly for the distributed implementation of S_{Ψ});
– relying on a communication-based distribution strategy (i.e. a protocol) and then using the knowledge obtained through static analysis in order to reduce the need for communication.

Such an approach may facilitate the construction of distributed controllers achieving reasonable performance at the implementation level (e.g., in terms of progress, or number of messages exchanged), and this for a larger class of systems.

The paper is structured as follows. In Section 2, we give an overview of approaches proposed for various application domains to achieve distributed control or distributed implementations based on global specifications. In Section 3, we formalize the centralized control problem using (constrained) Petri nets. In Section 4, we express the problems related to the distribution of constrained Petri nets. We discuss the use of static knowledge from the centralized specification and of communication for achieving distributed knowledge. In Section 5, we sketch a knowledge-based representation of the distributed algorithm presented in [11] and discuss the potential of optimization.

2 Related Work

The problem of deriving distributed implementations from global specifications and that of distributed control to enforce a global invariant have been studied intensively since the eighties. We provide here an overview of some important results in these domains, organized around three topics: The distributed implementation of synchronous languages, the derivation of protocols from specifications, and distributed control, with an emphasis on knowledge-based approaches. Other closely related areas are test and analysis of distributed implementations which we do not discuss here.

Distributed Implementation of Synchronous Languages. In synchronous languages [5], global specifications are given as a set of concurrent interacting components with local data, similar to what is done in hardware description languages. However, in the synchronous context, classical compilers do not generate a parallel implementation but a unique sequential program which may be executed on simple hardware platforms without any middleware. The need to distribute such a specification stems from the fact that the physical hardware is actually distributed, and different components (as defined at the specification level) run on different hardware units. The specification generally represents some real-time control system with rather tight synchronization constraints.

To derive distributed implementations in this context, the control flow is driven by (local) clocks, and the data exchanged between locations are continuous flows. Most synchronous languages define Kahn networks [13], that is, deterministic specifications where each variable is written at most once in each computation step, and no circular dependencies exist amongst them. Therefore, according to [13], achieving a correct distributed execution is straightforward on a platform with communication through unbounded FIFO buffers. Such implementations are reliable, but uninteresting in the context of real-time systems. Interesting applications require communication and computation time to be bounded, such that bounded buffers are sufficient and real-time constraints can be guaranteed [6].

Protocol Derivation. In the domain of telecommunications, automatic protocol generation from a global service specification was a hot topic in the eighties. Actions in the global service specifications may represent (oriented) data transfer or genuine synchronizations belonging to more than one physical location. Besides, specifications often feature some non-determinism which represents detail abstraction of how decisions are taken as well as some degree of openness of the design to be resolved later. However, a closer look reveals that-non determinism is often used to mimic concurrency.

There has been a huge amount of work in the eighties on communicating finite state machines [24,18] or formal specification languages such as LOTOS [21,14] to mention just a few. Some works propose methods for Petri nets with data transfer (through registers). For example, [23] presents an algorithm for generating, starting from a Petri net, a message passing protocol by means of a set of message synthesis rules. This line of work supposes that the control over interactions (that is, who is the initiator of an interaction) is solved a priori based on the direction of data flow, and that conflicts can always be solved locally. A more general method dealing also with conflicts and multi-party synchronizations has been proposed by Bagrodia [1], taken up in [17] for defining the α-*core* protocol — but note that it is not automatically derived.

Almost all the above-mentioned approaches aim at maximal progress, which is only one among many possible refinement relations. Besides, only few papers provide correctness proofs. When these are given, they are written in an ad hoc manner to establish the existence of a (bi-)simulation relation based on the introduced concepts. Knowledge-based reasoning offers exactly the right formalism to perform such proofs and therefore revisiting protocol derivation using knowledge seems a promising idea.

Distributed Control and its Knowledge-Based Formulations. The problem of achieving distributed control of a plant with respect to a global specification is closely related to the distribution problem. Here, for a given set of possible next actions supported by the plant, the aim is to allow in a distributed fashion one or more of them to be executed, using a set of controllers with some partial vision on the present situation. This requires – as before – to find some enabled actions, and when there are more than one, to detect whether there is conflict, making a choice amongst enabled actions if needed. In this specific context, instead of initiating the local part of a global action, local controllers provide a judgment on whether or not they propose the action for execution (see e.g. [16,22]).

We are here particularly interested in the methods presented in [12,20,19] where a knowledge-based presentation of the distributed control problem is proposed for systems, even if without the possibility of actual conflict situations. In [20] only *negative knowledge* is used: a local controller *knows* locally when a cannot be executed if this is due to its local protocol, and in order to forbid a at least one local protocol must do so. In [12] the notion of *knowledge-based protocol* is proposed as a means for representing protocol specifications abstractly: the local action a_i of P_i is *enabled* if P_i *knows* this fact in its present state. Obviously, knowledge depends on the global system and not just on the local state. Constructing a distributed protocol consists therefore in transforming this *external knowledge* into an *acquired knowledge* which can be locally exploited.

This knowledge-based approach has been taken up and generalized in [2,9,3,10,15,4] by suggesting the use of model checking for calculating knowledge properties in local states. This is done for global service specifications given in terms of Petri net like formalisms. The objective there is not maximal progress but deadlock preservation and minimization of communication. A problem is that there may not exist enough knowledge to take local decisions. In [9,3] it is therefore proposed to enrich the specification with some additional transitions representing temporary synchronizations. This work relies on some distributed protocol such as α-core for achieving a real distributed implementation, and for resolving conflicts. One limitation so far is that it is based on a somewhat unrealistic notion of locality, which makes the basic fireability condition of joint transitions local, but is not consistent with the underlying protocol. In [4] such a priori knowledge computation is used to avoid actual conflicts by eliminating some alternatives statically.

Our goal here is to integrate these approaches with the underlying protocol. In particular, we aim at simplifying the α-core protocol that was extended in [11] to handle global priorities. We would like to propose a knowledge-based formulation of this algorithm, in order to make it easier to verify and adapt to different notions of correctness and to different platforms.

3 Centralized Controlled Specifications

In order to build a distributed implementation of a constrained system, we proceed step by step, starting with a global (centralized) system specification, a constraint to be enforced and some properties of interest, until an executable implementation for a given distributed platform has been obtained. Our approach is particularly useful when the global (possibly constrained) specification can be checked — with reasonable complexity — for satisfaction of global properties whereas this is much more difficult or even infeasible to obtain on the distributed implementation.

3.1 Petri Nets

We use one-safe Petri nets as a convenient generic formalism to represent global specifications as well as distributed implementations. To simplify presentation, we suppose here that global specifications contain no data. We rather focus on potentially complex control structures: we consider symmetric multi-party synchronizations, allow arbitrary conflict situations, and specify global constraints such as priorities between transitions.

Definition 1. *A* Petri net N *is a tuple* (P, T, E, s_0) *where:*

- P *is a finite set of* places. *The set of* states *(markings) is defined as* $S = 2^P$.
- T *is a finite set of* transitions.
- $E \subseteq (P \times T) \cup (T \times P)$ *is a bipartite relation between places and transitions.*
- $s_0 \subseteq 2^P$ *is an* initial state *(initial marking).*

For a transition $t \in T$, we define the set of *input places* ${}^\bullet t$ as $\{p \in P | (p,\, t) \in E\}$, and the set of *output places* $t^\bullet$ as $\{p \in P | (t,\, p) \in E\}$.

Definition 2. *A transition t is called* enabled *in a state* s *if* ${}^\bullet t \subseteq s$ *and* $(t^\bullet \backslash {}^\bullet t) \cap s = \emptyset$. *We denote the fact that t is enabled from* s *by* $s[t\rangle$. *An* event, *corresponding to the firing t, leads from state* s *to state* s', *which is denoted by* $s[t\rangle s'$, *when* t *is enabled in* s *and* $s' = (s \backslash {}^\bullet t) \cup t^\bullet$.

A state s is in *deadlock* if there is no enabled transition from it.

Definition 3. *Two transitions* t_1 *and* t_2 *are* independent *if* $({}^\bullet t_1 \cup t_1{}^\bullet) \cap ({}^\bullet t_2 \cup t_2{}^\bullet) = \emptyset$.

That is, transitions are independent or *concurrent* if they do not influence each other.

We use the Petri net of Figure 1 as a running example. As usually, transitions are represented as segments, places as circles, and the relation E as a set of arrows from transitions to places and from places to transitions. The Petri net has places named p_i and transitions named $a, b, \ldots, g$. We represent a state s by putting *tokens* inside the places of s. In the example, the depicted initial state s_0 is $\{p_1, p_2, p_5\}$. The transitions enabled in s_0 are a and b. Note that in our case there cannot be more than one token in any place. Indeed, according to Definition 2, a transition t is enabled in a state s only if (after removing the tokens from the input places of t) there is no token in any of the output places of t. That is, using usual vocabulary for Petri nets, our Petri nets are *one-safe* by construction.

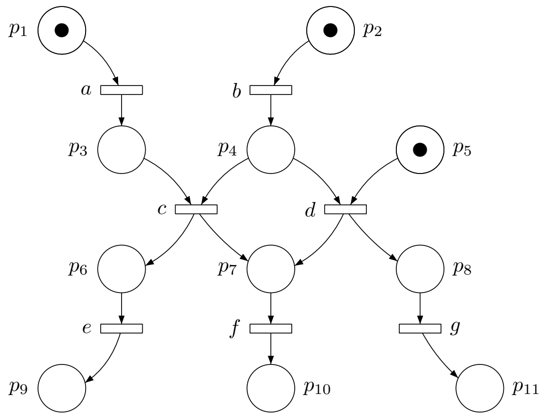


Fig. 1. A Petri net with initial state $\{p_1, p_2, p_5\}$

Definition 4. *An* event trace *is a maximal sequence of events $s_0[t_1\rangle s_1 \cdot s_1[t_2\rangle s_2 \cdot \ldots$ with s_0 the initial state of the Petri net, and any two consecutive events share their final, respectively initial state in the obvious manner.*

We denote the set of event traces of a Petri net N by $exec(N)$. The set of prefixes of the event traces in a set X is denoted by $pref(X)$. A state is *reachable* in N if it appears in at least one event trace of N. Our running example has 16 reachable states, for instance $\{p_3, p_7, p_{11}\}$. We denote the set of reachable states of N by $reach(N)$.

3.2 Centralized Control

On top of the Petri net specification, we want to enforce some global (safety) constraint.

Definition 5. *Given a Petri net $N = (P, T, E, s_0)$ with set of states S, a* control safety constraint *$\Psi \subseteq S \times T$ defines for each state s the set of transitions allowed in s.*

Such a constraint Ψ has the potential effect of forbidding some transitions allowed in N. We use this type of safety constraints as (1) they are trivially enforceable, and (2) any enforceable safety constraint can be transformed into a constraint of this form.

Definition 6. *An event trace of N constrained by Ψ, called* constrained event trace*, is a maximal prefix $s_0[t_1\rangle s_1 \cdot s_1[t_2\rangle s_2 \cdot \ldots$ of an event trace of N such that for each event $s[t\rangle s'$ in the sequence, (s, t) is in Ψ.*

We denote the set of constrained event traces of N with respect to Ψ by $exec(N, \Psi)$ and the set of reachable states under constraint Ψ (that is, states that appear in at least constrained event trace) as $reach(N, \Psi)$. Note that these constrained event traces express the set of allowed behaviors of the constrained system but do not describe how to enforce such behaviors.

As a running example of a constraint, we choose priority orders, similar to [2,9,3,10,15], used to discriminate between simultaneously enabled transitions in N. Note that these transitions may be concurrent or not.

Definition 7. *A* priority order $\ll$ *is a partial order relation on the transitions T of N. In a state s, transition t is said to be* maximally enabled *if it is enabled, and in s there is no enabled transition t' with higher priority, that is, such that $t \ll t'$.*

A priority order $\ll$ can easily be encoded as a safety constraint $\Psi_{\ll}$ defined as the set of pairs $(s, t) \in S \times T$ such that t is maximally enabled in s.

Example 1. Consider Petri net N of Figure 1 constrained by the priority order $\ll$ defined by $\{a \ll b, e \ll f, f \ll g\}$. The state $\{p_2, p_3, p_5\}$ is in $reach(N)$ but not in $reach(N, \Psi_{\ll})$, because in the initial state a may not be fired before b.

We formally represent the control imposed on N to enforce Ψ, by extending N with variables, additional enabling conditions on transitions, and data transformations associated with transitions so as to obtain an *extended Petri net* [8] N' whose event traces are exactly $exec(N, \Psi)$.

Definition 8. *An* extended *Petri net N' consists of*

- *a Petri net $N = (P, T, E, s_0)$ called the* underlying Petri net *of N';*
- *a finite set of variables V with given initial values $\mathcal{V}_0$;*
- *for each transition $t \in T$,*
  - *an enabling condition en_t, i.e., a predicate on the variables in V*
  - *some transformation predicate f_t on variables in V*

Definition 9. *An* execution *of an extended Petri net is a maximal sequence of the form $(s_0, \mathcal{V}_0) \cdot t_1 \cdot (s_1, \mathcal{V}_1) \cdot t_2 \cdot (s_2, \mathcal{V}_2) \ldots$ such that for all $i \geq 0$ we have: $s_i[t_{i+1}\rangle s_{i+1}$, $\mathcal{V}_i \models en_{t_i}$ and $\mathcal{V}_{i+1} = f_{t_i}(\mathcal{V}_i)$. The corresponding* event trace *is obtained by projecting out the variables of the execution and representing the sequence as a sequence of events.*

As previously for non extended Petri nets, we denote the set of event traces of N' by $exec(N')$. Note that an event trace of N' is a prefix of an event trace of the underlying Petri net as N' can only restrict the event traces of its underlying Petri net N, not generate new ones. It may however introduce deadlocks, and more generally, affect the progress properties of N. Formally, this means that $exec(N') \subseteq pref(exec(N))$.

Coming back to our control problem, we now define one possible criterion for defining the notion of correctness of an implementation of a constrained Petri net. We then show how a Petri net N can be extended to enforce a constraint Ψ. One of the challenges in the remainder of this paper is to distribute this controller.

Definition 10. *An* extended *Petri net N'* implements *a Petri net N constrained by Ψ if $exec(N') \subseteq exec(N, \Psi)$.*

Note that this is a quite restricted definition of correct implementation as it forces the use of the same state structure, but it is sufficient for the illustrative purpose of this paper. More importantly, this definition forbids N' to introduce new deadlocks compared to N constrained by Ψ. It does not require any stronger progress, meaning that the only properties which are preserved by this definition are safety and deadlock-freedom. Quite clearly, other definitions are possible here.

Proposition 1. *Given a Petri net N and a constraint Ψ on N, the extended Petri net $N' = (P, T, E, s_0, V, \{en_t\}_{t \in T}, \{f_t\}_{t \in T})$ where*

- *the underlying Petri net is $N = (P, T, E, s_0)$*
- *$V = \{v\}$ where v encodes the state of N*
- *for each transition $t \in T$, en_t holds if and only if $(v, t) \in \Psi$*
- *for each transition $t \in$, f_t updates v to the new state*

is such that N' implements N constrained by Ψ.

We say that $\{V, \{en_t\}_{t \in T}, \{f_t\}_{t \in T}\}$ defines a controller *for N enforcing Ψ and we call the event traces of N'* controlled *event traces of N.*

Proof. Any event trace of N' is clearly also a constrained event trace with respect to Ψ. In fact, in this centralized context we even have the stronger property that $exec(N') = exec(N, \Psi)$. Indeed, our controller allows all transitions permitted by Ψ.

We already mentioned the need to verify global properties of interest at a high level of abstraction whenever possible so as to avoid the state-space explosion problem — remember that our goal is to provide a distributed implementation of the constrained Petri net. As the event traces of (N, Ψ) are also event traces of N, all the safety properties proven on N hold on (N, Ψ). For progress however, the situation is different, as Ψ may block some transitions allowed by the original Petri net. This means that one must either prove progress directly on the constrained system, or use a correct-by-construction approach. This issue will reappear for the relation between the event traces of the distributed implementation and those in $exec(N, \Psi)$.

4 Distributed Implementations and Control

We want to use Petri nets to specify and analyze the global behavior of a *distributed* system. In practice, the system consists of a set of concurrently executing and temporarily synchronizing processes. Such a distributed implementation supposes a platform in which each process has access only to its *local* view of the system execution but may communicate with other processes using some mechanisms provided by the platform.

In this section, we proceed as follows: we first focus on the definitions related to the implementation of distributed systems with constraints. Then, we formalize our solution for controlling such systems using knowledge and communication.

4.1 Distributed Petri Nets

First, we define a Petri net as a distributed system of processes. There are several options for defining the notion of process in Petri nets: we choose to consider place sets.

Definition 11. *A* process π *of a Petri net N is a subset of the places of N (i.e., $\pi \subseteq P$) such that there is always exactly one token in π.*

Definition 12. *A* distributed Petri net *is a pair (N, Π) where N is a Petri net as in Definition 1 and Π a set of processes of N defining a partition of the set of places of N.*

From now on, we assume a distributed Petri net (N, Π). Figure 2 illustrates as an example a possible distribution of the Petri net of Figure 1. In the sequel, we keep the priority order of Example 1. For each transition t, we denote $proc(t)$ the set of processes which have at least one place in ${}^\bullet t$. Note that, because we consider only sequential processes here, this set is exactly the set of processes which have at least one place in $t^\bullet$, and furthermore, the processes in $proc(t)$ have exactly one place in ${}^\bullet t$ and $t^\bullet$, denoted respectively ${}^\bullet t_\pi$ and $t^\bullet{}_\pi$ (we reuse this notation for corresponding singletons).

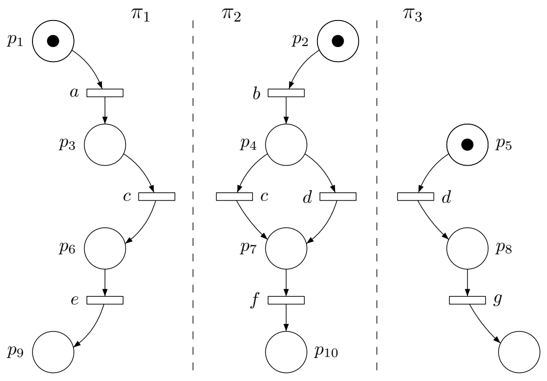


Fig. 2. A distributed Petri net with priority order $\{a \ll b, e \ll f, f \ll g\}$

Definition 13. *The* local state *of a process π in a (global) state $s \in S$ is defined as $s_{|\pi} = s \cap \pi$. A local state s_π of π is part of a global state $s \in S$ if and only if $s_{|\pi} = s_\pi$.*

That is, the local state of a process π in a global state s is the projection of s onto the places of π. It describes what π can see based on its limited view of the system.

We now define the notations related to the local execution of π which we define as the execution depending only the local view of π. E.g., a transition t may not be enabled in a state s but still be enabled in a local $s_{|\pi}$.

Definition 14. *A* local event *for π, corresponding to the* firing *of a transition t such that $\pi \in proc(t)$, leads from local state s_π to local state s'_π, which is denoted by $s_\pi[t\rangle s'_\pi$, when t is* locally enabled *in s_π — that is, s_π has one token in ${}^\bullet t_\pi$ (and therefore no token in $t^\bullet{}_\pi$) — and $s'_\pi = (s_\pi \backslash {}^\bullet t_\pi) \cup t^\bullet{}_\pi$. A* local event trace *of π is a maximal sequence of local events $s_0^\pi[t_1\rangle s_1^\pi \cdot s_1^\pi[t_2\rangle s_2^\pi \cdot \ldots$ as before.*

We denote local enabledness of t in s_π by $[t^\pi\rangle$. Now we define a distributed event trace as an arbitrary interleaving of local event traces which we represent here simply as the set of local traces.

Definition 15. *A* distributed event trace *is a tuple $(\sigma^{\pi_1} \ldots \sigma^{\pi_n})$ of* local event traces *which contains one local event trace per process in Π, and possibly a precedence relation $\prec$ relating events of different processes. $\prec$ restricts the allowed interleaved event traces σ*[1].

[1] Here, we say in the implementation relations defined in the next section how $\prec$ is defined.

Clearly, even for a very relaxed notion of *correct distributed implementation*, the distributed Petri net obtained by duplicating transitions shared by several processes does in general not behave according to the (centralized) Petri net N, even without considering the constraint Ψ. We need to control the system, in order to enforce a correct implementation of both N and Ψ. Again, this is represented by an extended Petri net, where each process has a disjoint set of local variables.

Definition 16. *A* distributed controller *for a distributed Petri net* (N, Π) *is defined by a set of triples* $\{V^\pi, \{en_t^\pi\}_{t \in T}, \{f_t^\pi\}_{t \in T}\}$*, one for each process* π *in* Π.

Definition 17. *A* controlled distributed event trace *of* (N, Π) *by a controller is a tuple* $(\sigma^{\pi_1} \ldots \sigma^{\pi_n})$ *of* controlled local event traces *and a precedence relation* $\prec'$ *such that* $(\sigma^{\pi_1} \ldots \sigma^{\pi_n})$ *corresponds to a prefix of a local trace as in the centralized controlled case and* $\prec'$ *restricts* $\prec$.

Note that (N, Π) may be controlled by either a centralized or a distributed controller, but our goal is of course to find a distributed one. The challenge is then to find the right enabling conditions to control the distributed execution in order to ensure that the distributed system implements its global specification — according to the chosen criterion — while satisfying also the control invariant Ψ. This will be addressed in Section 4.3. But before that, let us discuss possible options to define what it means for a distributed Petri net to *implement* a (centralized) Petri net.

4.2 Correctness Criteria for Distributed Implementation of Petri Nets

Let us consider first a very weak implementation relation which guarantees only sequential consistency, that is, no relation $\prec$ relating events of different processes:

(1) *transition correctness* which ensures that the local order of transitions is preserved.
(2) *atomicity* which requires, in case of a conflict situation, that all involved processes take the same decision.

All the upcoming definitions apply not only to distributed Petri nets but also to *controlled* distributed Petri nets (traces in the definitions are then controlled).

Definition 18. *Given a distributed Petri net* (N, Π)*, the projection of a trace* σ *of* $exec(N)$ *(or* $exec(N, \Psi)$ *if the system is constrained) on a process* $\pi \in \Pi$ *is a local trace obtained from* σ *by keeping only events which involve transitions* t *such* $\pi \in proc(t)$ *and projecting all states* s *onto the corresponding local state* $s_{|\pi}$*. That is, we through away all non local ordering constraints.*

Definition 19. *A distributed Petri net* (N, Π) implements N *constrained by* Ψ *with respect to* $\preceq_{noSync}$*, if for every distributed event trace* $(\sigma_1, \ldots \sigma_n)$*, there exists a (centralized) controlled event trace* $\sigma \in exec(N, \Psi)$ *such that for all processes* π_i *the projection of* σ *on* π_i *matches the corresponding local trace* σ_i.

This relation is not necessarily implementable, as it may require unbounded buffering. But usually, an implementation includes also two other types of constraints:

(3) *synchronization* constraints, which restrict the allowed interleavings of local events. For example, when synchronizations represent an asymmetric situation, like a write and corresponding reads, on may restrict the order amongst local events corresponding to the same (global) event. But we do not necessarily want to impose such *causality* constraints a priori, in order to be able to model out of order executions, prefetches, which "apparently violate causality".
(4) *progress* constraints, which range from absence of global or local deadlock to maximal progress, meaning that for each event trace of N there must exist a distributed event trace of (N, Π).

We now present the implementation relation for the fully synchronized case.

Definition 20. *A distributed Petri net* (N, Π) implements *a Petri net* N *constrained by* Ψ *according to* $\preceq_{fullSync}$, *if (1) the condition of Definition 19 is satisfied, and (2) all interleaved event traces* σ *of* (N, Π) *satisfy the following* synchronization condition*: whenever for two non-independent events* a, b, a *occurs before* b *in the global trace* σ^N *for* σ, *then all events* a_π *in* σ *occur before all events* $b_{\pi'}$.

This is a very strong correctness criterion, which in practice is rarely necessary, and rarely implemented this way. We consider instead another implementation relation $\preceq$ that is widely used in protocols, e.g. for α-core and [11]. This relation requires processes to synchronize before the execution of a transition, but not on termination.

Definition 21. $\preceq$ *is defined as* $\preceq_{fullSync}$ *except that condition (2) only requires that all events* a_π *in* σ *occur before all events* $b_{\pi'}$ *for the processes* π, π' contributing to b.

It is essential to note here that for each of these relations, a correct implementation may reach states which are *not* in $reach(N, \Psi)$. For example, in the fully synchronized case, such states correspond to intermediate states during the firing of a transition. Note that, e.g. in pipelined executions, the distributed implementation may never reach any state of the centralized execution, but all implementation relations require that by *executing the transitions "lagging behind"* a state of the original Petri net is reached. A fact that is used in the domain of hardware verification.

4.3 Using Knowledge and Communication for Distributed Control

We now focus on the question of how to build a correct distributed implementation of the constrained Petri net. Remember that we need, for each process, a set of variables, and for each transition an enabling condition and an update function. Consider first the relation $\preceq_{fullSync}$, which is the closest to the centralized Petri net. To simplify notation, we suppose that the Petri net has no loop (if needed it can be unfolded to an infinite state Petri net) so that each transition may be fired at most once. This allows us to define a property $done_t^\pi$ that holds exactly when a process π has already fired transition t.

We can define a centralized controller for (N, Π) enforcing Ψ by

- $V = \{v_\pi\}_{v \in \Pi}$ where v_π encodes the local state of process π;
- for each transition t, en_t^π is as defined below;
- for each transition t, f_t^π updates v_π to the new local state.

A process π may locally fire a transition t in a (global) state s if and only if s satisfies the enabling condition en_t^π defined as the conjunction of the following properties:

1. t is either globally enabled or already partially executed, that is
$$ready_t = \forall \pi' \in proc(t) \,.\, ([t^{\pi'}\rangle \vee done_t^{\pi'})$$
2. the transitions t' executed previously in π are all terminated (in all processes)
$$\forall t'.(done_{t'}^{\pi} \implies done_{t'}) \; where \; done_{t'} = \forall \pi' \in t' \,.\, done_{t'}^{\pi'}$$
3. t has maximal priority[2] — denoted $\max_t$
4. t has no unresolved conflict, meaning that all processes involved in t — i.e. $\pi \notin proc(t)$ — will indeed fire t if they have not already done so. We express that there is no unresolved conflict by a property $selected_t$ which expresses this conflict resolution by guaranteeing that for any transition t' potentially in conflict with t, $\neg selected_{t'}$ holds.

These conditions guarantee the properties required by $\preceq_{fullSync}$. The first and the second one guarantee transition correctness (only legal transitions can be executed), the third one guarantees atomicity, and the synchronization constraint is guaranteed by the first (which guarantees the rendezvous on the input state) and the second (which guarantees the rendezvous on the output state).

The enabling condition en_t^π for the loosely synchronized relation $\preceq$ is the same, except that the second condition may be dropped, as $\preceq$ does precisely not require a synchronization on the termination of a transition.

Notice, that these enabling conditions for local transitions depend all on the global state and therefore this controller is not distributed. It requires visibility on the entire state for deciding whether t can be executed. We want to use the notion of *knowledge* [7] to solve this issue. The *knowledge* of a process π in a local state s_π is the set of reachable states s which project onto s_π, i.e., such that $s|_\pi = s_\pi$.

Definition 22. *A process* knows *a property φ is a local state s_π, denoted $s_\pi \models K_\pi \varphi$ if and only if φ holds in all the reachable (global) states s such that $s_{|\pi} = s_\pi$.*

For example, in p_6 of Figure 2, process π_1 knows that π_3 is in local state p_5 as c and d may not both have been fired. We sometimes denote this by p_6 knows p_5.

Regarding our local enabling conditions, this means that processes must *know* that the global enabling condition holds, i.e., that it must hold in all global states the local state s_π cannot distinguish. In other words, by replacing en_t^π by $K_\pi en_t^\pi$ we obtain a distributed controller.

A second important point is, that calculating knowledge at the level of the centralized Petri net is a priori not sufficient. The enabling conditions en_t^π defined earlier must be known at the level of the *distributed* system. That is, in Definition 22, the set of reachable states is that of the distributed system and must include interleavings. As it is obviously much more interesting to calculate the knowledge of N, we study next the preservation of knowledge properties for N in the distributed implementation.

[2] In the general case, t is enabled with respect to Ψ.

Exploiting the Knowledge of the Centralized Petri Net. We address here the following question: which of the knowledge properties that we have computed on the centralized Petri net N can be exploited in the enabling conditions of local processes? To answer this question, we can use an obvious closure property of knowledge sets.

Proposition 2. *If a state s is in the knowledge set of a local state s_π, then so are all states reachable via the execution of transitions t in which π is not involved.*

In a centralized execution, the partners of a transition t move jointly, and therefore the states before and after t in processes π' in $proc(t)$ are in disjoint knowledge sets. On the other hand, in a decentralized computation, the participants in t execute their local t independently. This means that the knowledge set of ${}^\bullet t_\pi$ (and $t^\bullet{}_\pi$) contains ${}^\bullet t_{\pi'}$ *and* $t^\bullet{}_{\pi'}$ of such processes. Thus, the looser the implementation relation (that is, the more desynchronized local processes may be), the weaker the knowledge that a process has in a given local state. Note however that looser implementation relations require weaker enabling conditions on local transitions.

Let us have a look at the example of Figure 2, to see whether there is some useful knowledge that is preserved, for example when the implementation relation is $\preceq$. In the centralized execution p_3 *knows* p_4 (b has higher priority than a). Thus, it knows en_c as c is not dominated by any transition of higher priority. In a distributed execution according to $\preceq$, p_3 only *knows* $p_4 \vee p_7 \vee p_{10}$ which — together with the information $selected_c$ — still implies en_c^1. As we have already stated, $selected_c$ is not a knowledge that can be present in N in a conflict situation, the decentralized implementation needs some additional decision mechanism. Besides, as $f \ll g$, in N, p_7 *knows* $\neg p_8$ which is sufficient to execute f. This holds also in an implementation according to $\preceq$.

Indeed, we can characterize the transformation of knowledge of the centralized system into knowledge of the distributed system as follows.

Proposition 3. *Whenever a local state s_π knows a local predicate $p_{\pi'}$ concerning process π' in the Petri net N, then it knows in (any of) the distributed semantics the property $past[\neg sync(\pi,\pi')](p_{\pi'}) \vee AF[\neg sync(\pi,\pi')](p_{\pi'})$, which is a CTL formula expressing the fact that $p_{\pi'}$ has been true in a past after the last synchronization with π, now, or in the future before the next synchronization with π.*

The *last* and *next* synchronizations mentioned above depend on the synchronization points defined by the preorder[3]. But, one can rely on the synchronization points imposed by the preorder only for synchronization points that lie strictly in the past or strictly in the future, not for *realizing* a rendezvous *now*. Only a property that is *closed* in this sense with respect to past and future is preserved. These are in particular properties stating that a neighbor cannot be, or must be, before or beyond a certain point. The enabledness condition for implementation relations $\preceq$, and $\preceq_{fullSync}$ are not of this nature, and therefore *not* preserved by construction.

The obvious conclusion is that, although useful, knowledge is in general not sufficient to achieve a distributed implementation without additional communication. In the next section we study how knowledge can be used jointly with communication.

[3] They obviously also depend on the additional synchronization points actually achieved in the distributed implementation under study.

Increasing Knowledge by Communication. Let us first note that communication is in principle only required for achieving progress. Otherwise, eliminating transitions for which we do not have sufficient knowledge is a safe way of proceeding, and moreover this increases knowledge. Of course, in most cases this is not an acceptable option. Our goal here is not to determine a particular, optimal communication strategy but rather to provide some insight into how knowledge can be used to analyze and optimize for example a *given* distribution protocol such as α-core or the one in [11].

Remember that a process π needs in a local state s_π the knowledge of the following enabling conditions to execute transition t: (1) $ready_t$, i.e. global readiness (2) max_t, i.e. maximal priority (3) $selected_t$, i.e. absence of unresolved conflict.

A local state s_π does in general not *know* the conjunction of these properties, and communication via the distributed platform can be used towards it. Semantically, the information conveyed by the communication reduces the set of global states consistent with s_π (meaning that $s_{|\pi} = s_\pi$). And fewer states means stronger knowledge.

Knowledge concerning readiness and maximal priority can be obtained by collecting information on local enabledness in other processes. The platform will (asynchronously) interact with processes π and π' to inform π' that in π "t was locally enabled" or t' *guaranteed to be not enabled before* t'' in a previous global state that is consistent with the local state of π'. Now, local enabledness is not a stable property. It may be the case that when π' synchronizes with the platform to obtain that information, π is no longer enabled. This, however, is taken care for in the knowledge property. In other words, such communication delivers additional information, which allows eliminating "states of the past" from the knowledge set of s_π. As already stated earlier, knowledge properties are stable with respect to invisible evolution, and mean π' *is already beyond that point.* On the other hand, the protocol may force π' to *wait* in certain situations, which provides knowledge of the form π' *is not yet beyond that other point.* This allows synchronization.

Example 2. In local state $\{p_1\}$ of our running example of Figure 2, π_1 cannot fire a because its local state is consistent with $\{p_1, p_2, p_3\}$ in which b is enabled and therefore a may not have maximal priority. On the other hand, if π_1 knows that c has been locally enabled in the past of π_2, then $\{p_1, p_2, p_3\}$ is no longer a possible current state and therefore π_1 knows that it may fire a.

Note however, that the third part of the enabling condition en_t^π, the condition $selected_t$ is satisfied in the original Petri net only if there is a unique enabled transition in s, which is then the selected one. The choice between conflicting transitions requires additional information which may be of static nature, that is, consist in modifying N or dynamic, that is, the protocol needs to include a distributed arbitration protocol. The algorithms of [17,11] include such arbitration protocols. Most other solutions from the literature simply forbid conflicts.

5 Discussion Based on the Protocol of [11]

Let us now discuss the added value of the reasoning presented in the previous sections on an example. In [11], an algorithm is presented which builds a distributed implementation of a prioritized specification for systems with binary synchronizations. It is inspired by α-core but differs from it by the fact that it handles specifications with (global) priorities and implements a less static conflict resolution. In both algorithms, the platform is assumed to ensure reliable and order-preserving transmission of messages. The precise organization of the protocol of [11] is beyond the scope of this paper, but an abstract general view of the main steps of the communication phase is shown in Figure 3. This protocol takes place in each local state of each process, that is, all the states represented in the diagram correspond to the same place in the centralized Petri net. The transition in the diagram labeled 7 corresponds to the start of the execution of a local transition (which takes place in the $Busy$ state). Other transitions represent sending and receiving of messages expressing information about local enabledness of transitions, as well as commitment of a process to a given transition t — the last step before achieving $selected_t$, unless the other process rejects t.

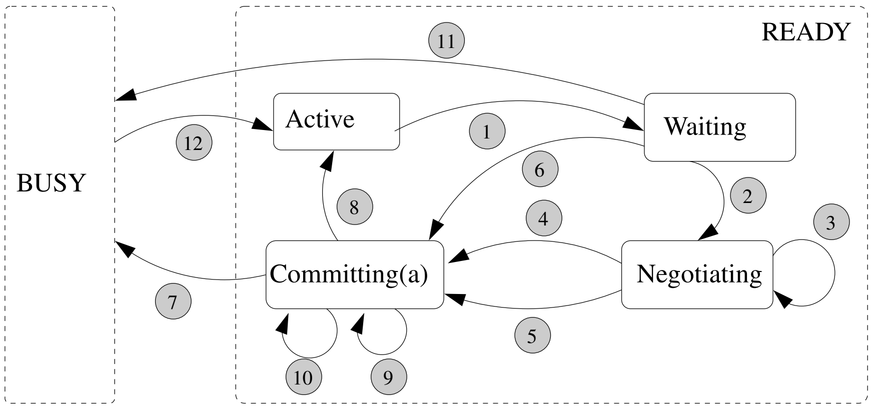


Fig. 3. State diagram of the algorithm presented in [11]

We can use the results of Section 4 in two ways here. First, we now have a generic formal support for proving the correctness of the algorithm under consideration. Indeed, transition correctness, atomicity and synchronization as defined in Section 4.3 are satisfied if and only if a process π may take transition 7, that is, fire locally a transition t of the original Petri net, only when it has the required *knowledge* for it. For example, $selected_t$ holds in that case because both processes π and π' have committed to it. The protocol obliges π' to wait for the consent of π, and π therefore *knows* that π' cannot fire any other transition than t. In other words, in all the global states consistent with the local view of π in this local state, $selected_t$ holds.

Second, once formalized the knowledge properties associated with each local state, we can use them in combination with the properties obtained by static analysis of the centralized Petri net. That is, for a local state s_π of π, all states of the associated communication protocol are enriched with (preserved) local knowledge of π in s_π. Based on this, π may not have to wait for all messages to arrive before progressing, as it now has enough knowledge to fire a transition without them. In addition, if messages are clearly identified as questions and answers — as is often the case in such protocols — then π may in such case omit some questions messages as it does not require them. However, this forbids the analysis of progress properties to rely on question messages.

Such a clear separation between the generic protocol and its implementation for a given centralized Petri net seems promising, as it is scalable (the distributed system as a whole is never analyzed) and still understandable: centralized Petri net and protocol are analyzed separately, then used together in a correct-by-construction manner.

6 Conclusion

In this paper, we have discussed a knowledge-based representation of the distribution problem which then can be used for either deriving distributed implementations automatically from global specifications on which a given constraint is to be enforced, or for the optimization of existing protocols by exploiting pre-calculated knowledge. Our intention was not to provide an exhaustive treatment of this topic — which has been already studied quite extensively in the past, for various needs and from different perspectives. We hope however, to have illustrated that such a knowledge-based approach provides the right level of abstraction for solving the distribution problem depending on the notion of refinement required and the distributed platform at hand.

We have discussed different types of knowledge properties that may be required to take a decision globally or locally and how such knowledge can be obtained statically, or dynamically, or by mixing both approaches. In particular, we have illustrated that expressing refinement and local enabledness as a conjunction of smaller properties has several advantages, namely that of breaking down the verification of correctness.

What we did not consider at all in this paper, is dataflow, which introduces additional ordering, and how this may fit into the proposed framework. Similarly, we have not mentioned timed specifications, although it happens frequently that the distributed a system must satisfy time constraints under some timing assumptions — where the first add additional order constraints and the second may be useful for achieving them locally.

References

1. Bagrodia, R.: Process synchronization: Design and performance evaluation of distributed algorithms. IEEE Trans. Software Eng. 15(9), 1053–1065 (1989)
2. Basu, A., Bensalem, S., Peled, D., Sifakis, J.: Priority scheduling of distributed systems based on model checking. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 79–93. Springer, Heidelberg (2009)
3. Bensalem, S., Bozga, M., Graf, S., Peled, D., Quinton, S.: Methods for knowledge based controlling of distributed systems. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 52–66. Springer, Heidelberg (2010)
4. Bensalem, S., Bozga, M., Quilbeuf, J., Sifakis, J.: Knowledge-based distributed conflict resolution for multiparty interactions and priorities. In: Giese, H., Rosu, G. (eds.) FORTE 2012 and FMOODS 2012. LNCS, vol. 7273, pp. 118–134. Springer, Heidelberg (2012)
5. Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., de Simone, R.: The synchronous languages twelve years later. Proceedings of the IEEE 91(1) (January 2003)

6. Caspi, P., Girault, A.: Execution of distributed reactive systems. In: Haridi, S., Ali, K., Magnusson, P. (eds.) Euro-Par 1995. LNCS, vol. 966, pp. 15–26. Springer, Heidelberg (1995)
7. Fagin, R., Halpern, J.Y., Vardi, M.Y., Moses, Y.: Reasoning about knowledge. MIT Press, Cambridge (1995)
8. Genrich, H.J., Lautenbach, K.: System modelling with high-level petri nets. Theor. Comput. Sci. 13, 109–136 (1981)
9. Graf, S., Peled, D., Quinton, S.: Achieving distributed control through model checking. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 396–409. Springer, Heidelberg (2010)
10. Graf, S., Peled, D., Quinton, S.: Monitoring distributed systems using knowledge. In: Bruni, R., Dingel, J. (eds.) FORTE 2011 and FMOODS 2011. LNCS, vol. 6722, pp. 183–197. Springer, Heidelberg (2011)
11. Ben Hafaiedh, I., Graf, S., Quinton, S.: Building distributed controllers for systems with priorities. J. Log. Algebr. Program. 80(3-5), 194–218 (2011)
12. Halpern, J.Y., Fagin, R.: Modelling knowledge and action in distributed systems. Distributed Computing 3(4), 159–177 (1989)
13. Kahn, G.: The semantics of simple language for parallel programming. In: IFIP Congress, pp. 471–475 (1974)
14. Kant, C., Higashino, T., von Bochmann, G.: Deriving protocol specifications from service specifications written in LOTOS. Distributed Computing 10(1), 29–47 (1996)
15. Katz, G., Peled, D., Schewe, S.: Synthesis of distributed control through knowledge accumulation. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 510–525. Springer, Heidelberg (2011)
16. Lin, F., Wonham, W.M.: Decentralized supervisory control of discrete-event systems. Inf. Sci. 44(3), 199–224 (1988)
17. Pérez, J.A., Corchuelo, R., Toro, M.: An order-based algorithm for implementing multiparty synchronization. Concurrency — Practice and Experience 16(12), 1173–1206 (2004)
18. Probert, R.L., Saleh, K.: Synthesis of communication protocols: Survey and assessment. IEEE Trans. Computers 40(4), 468–476 (1991)
19. Ricker, S.L., Rudie, K.: Knowledge is a terrible thing to waste: Using inference in discrete-event control problems. IEEE Trans. Automat. Contr. 52(3), 428–441 (2007)
20. Rudie, K., Ricker, S.L.: Know means no: Incorporating knowledge into discrete-event control systems. IEEE Transactions on Automatic Control 45(9), 1656–1668 (2000)
21. von Bochmann, G., Gotzhein, R.: Deriving protocol specifications from service specifications. In: Proceedings of SIGCOMM 1986, pp. 148–156. ACM (1986)
22. Wong, K.C., Wonham, W.M.: Modular control and coordination of discrete-event systems. Discrete Event Dynamic Systems 8(3), 247–297 (1998)
23. Yamaguchi, H., El-Fakih, K., von Bochmann, G., Higashino, T.: Deriving protocol specifications from service specifications written as predicate/transition-nets. Computer Networks 51(1), 258–284 (2007)
24. Zafiropulo, P., West, C.H., Rudin, H., Cowan, D.D., Brand, D.: Towards analyzing and synthesizingprotocols. IEEE Transactions on Communications 28(4), 651–661 (1980)

Automated Anonymity Verification of the ThreeBallot Voting System

Murat Moran*, James Heather, and Steve Schneider

University of Surrey, Guildford, UK
m.moran@surrey.ac.uk

Abstract. In recent years, a large number of secure voting protocols have been proposed in the literature. Often these protocols contain flaws, but because they are complex protocols, rigorous formal analysis has proven hard to come by.

Rivest's ThreeBallot voting system is important because it aims to provide security (voter anonymity and voter verifiability) without requiring cryptography. In this paper, we construct a CSP model of ThreeBallot, and use it to produce the first automated formal analysis of its anonymity property.

Along the way, we discover that one of the crucial assumptions under which ThreeBallot (and many other voting systems) operates—the Short Ballot Assumption—is highly ambiguous in the literature. We give various plausible precise interpretations, and discover that in each case, the interpretation either is unrealistically strong, or else fails to ensure anonymity. Therefore, we give a version of the Short Ballot Assumption for ThreeBallot that is realistic but still provides a guarantee of anonymity.

Keywords: Formal Methods, Voting Systems, FDR2, CSP, Anonymity, Automatic Verification, ThreeBallot.

1 Introduction

Recent years have seen a large number of end-to-end voting systems proposed in the literature [1, 2, 3, 4, 5]. Typically these systems aim to provide a proof of correctness of the election tally, but also some guarantee of privacy for the voter; and cryptography is usually employed to achieve these goals. Rivest's ThreeBallot voting system [5] is particularly interesting because it uses no cryptography, but nevertheless still aims to provide anonymity, integrity of the election, verifiability and incoercibility.

One of the most critical properties of voting systems is anonymity, which essentially requires that the link between voters and votes be broken. Anonymity is important for voter privacy as well as it is essential for preventing coercion and vote buying. This paper considers the anonymity property as it relates to the ThreeBallot voting system.

* Corresponding author. His work is sponsored by The Ministry of Education Republic of Turkey.

ThreeBallot relies heavily on the *short ballot assumption* (SBA) to assist in providing its anonymity guarantee. Roughly speaking, this assumption states that the information content of a ballot should be low. However, the phrasing of this assumption in the description of ThreeBallot is vague, and open to a number of radically different interpretations. We consider the various possibilities here. Some turn out to be unrealistically strong; some seem to be too weak to guarantee anonymity.

In the process, we construct a formal model of ThreeBallot in Communicating Sequential Processes (CSP) [6], and use the Failures-Divergences Refinement (FDR2) model checker [7] to produce an automated analysis of the model. Some other voting systems have been at least partially verified automatically against privacy-related properties (for example, Civitas [3] in [8] with hand-proofs, FOO [2] in [9] with a compiler, and Prêt à Voter [10] in [11]); but the ThreeBallot voting system has not yet been subjected to automated formal verification.

The paper is constructed as follows. In the remainder of this section, we give an outline of ThreeBallot, and discuss related work. In Section 2, we model ThreeBallot as a parallel composition of agents: voters, an authority, and a bulletin board. Then, using an anonymity definition given in [11], in Section 3.1 we analyse our model against an adversary who can observe all public channels. Initially, our model drops the SBA entirely, and we discover that FDR leads us to several attacks on vote anonymity. Section 3.2 then discusses the Short Ballot Assumption in its various guises, and shows that in each case the assumption is either too strong to be realistic or too weak to be secure; we then propose a different short ballot assumption that is both reasonable and demonstrably strong enough to provide anonymity. In the Section 3.3 we analyse the other versions of ThreeBallot, and demonstrate that with the modifications, ThreeBallot provides guaranteed anonymity. Finally, the Section 4 concludes this paper with a summary of findings and present limitations.

1.1 Voting with ThreeBallot

In this section, we briefly introduce the original ThreeBallot voting system and the short ballot assumption given by Rivest and Smith [12].

Voting in ThreeBallot proceeds as follows. Initially, the (authenticated) voter receives a multi-ballot form from a pollworker, which consists of three mini-ballot forms (see Table 1). The mini-ballots are all identical except for the IDs or serial numbers, located at the bottom of the mini-ballots. These serial numbers are all unique, and are not meaningful. In particular, there is no way of determining what mini-ballot serial numbers go together to make up a multi-ballot.

The voter fills two bubbles in total for the chosen candidate, and only one bubble for each other candidate. The completed multi-ballot is inserted into a checker, which confirms that it has been correctly completed.

Finally, the voter chooses one of the mini-ballots, and receives a duplicate of that mini-ballot as her receipt. She then separates the three mini-ballots, and casts them all individually into a ballot box.

Table 1. A ThreeBallot multi-ballot, filled as a vote for Alice

Alice	●	Alice	●	Alice	○
Bob	○	Bob	●	Bob	○
	56248		04578		31489

After the election, all mini-ballots are published on a web bulletin board, along with a list of everyone who voted. The voter may then verify that the mini-ballot for which she has a receipt appears unaltered on the bulletin board (BB); if it does not, she can produce the receipt as evidence of foul play. The number of votes for each candidate is counted as usual. However, as each voter fills in exactly two bubbles for the chosen candidate and one bubble for the other candidates, the number of voters is subtracted from each candidate's final tally to find the correct number of votes for each candidate. Since all the mini-ballots are posted on the bulletin board, the final tally can be verified by anyone.

ThreeBallot is claimed in [12] to be secure under the short ballot assumption (SBA). Rivest and Smith in [12] define the SBA as the assumption that

> the ballot is short—there are many more voters in an election than ways to fill out an individual ballot [...] It is reasonable to assume under the SBA that each possible ballot is likely to be cast by several voters.

The ambiguities arise from the terms "possible ballots" (mini-ballots or multi-ballots?) and "several voters" (how many?).

Looking elsewhere for clarification bears little fruit. According to [13] the SBA assumes that "the list of candidates on a ballot is short enough in order to guarantee security"; we read in [14] that "the length of the ballots must be kept small (possibly by splitting them into several parts)".

Because ThreeBallot is claimed to guarantee voter anonymity under the SBA, analysis of ThreeBallot is not possible without a clear and unambiguous reading of the assumption. We give here three possible interpretations; we will analyse ThreeBallot under each of these readings in Section 3.2.

In each case, the intention is that the assumption will be guaranteed probabilistically; that is, that the number of voters, candidates, etc., will be sufficient to ensure that the assumption is broken with only negligible probability. In what follows, serial numbers will be ignored; that is, two mini-ballots will be considered the same if they contain the same marks apart from the serial numbers.

Assumption 1 (SBA-multi). *Every possible multi-ballot will be cast at least once.*

The formulation of the SBA given in Assumption 1 requires that every possible way of completing a multi-ballot should be adopted by at least one voter. For small numbers of candidates, this is not implausible. For even moderate numbers, though, the assumption quickly becomes hard to stomach.

Note that once one has chosen a candidate, there are then exactly three ways of completing each row: for the chosen candidate's row, one must choose a bubble to leave empty, and for each other row, one must choose a bubble to fill. There are thus $c \cdot 3^c$ distinct multi-ballots, where c is the number of candidates standing in the election.

It is not feasible to calculate the number of voters required to make this reasonable, because it depends on the probability distribution of multi-ballots: voters do not cast multi-ballots randomly (one hopes). A full calculation would require a realistic model of how voters cast their ballots. However, the best case scenario is when voters cast their multi-ballots randomly; so by assuming a uniform distribution, we can determine a lower bound on the number of voters required.

With a uniform distribution, the expected number of voters needed to cover all possible multi-ballots is $n \cdot \sum_{i=1}^{n} \frac{1}{i}$ where $n = c \cdot 3^c$, the number of possible multi-ballots. For five candidates, this comes out at 9331 voters; for ten candidates, we need 8.1 million voters; for fifteen candidates, the number exceeds 4 billion.

For n possible multi-ballots, and a uniform distribution, we can calculate the number of voters required to ensure that the probability of covering every multi-ballot at least once exceeds a given threshold. Since the security of ThreeBallot relies on the SBA, we would need confidence that (the correct interpretation of) the SBA is satisfied; we can, therefore, for a given probability level, ask how many voters are required to give this level of confidence that the SBA will be satisfied.

For n multi-ballots, and v voters, the probability that the v voters will cover all of the n possibilities is

$$1 - \sum_{j=1}^{n-1} (-1)^{j+1} \binom{n}{j} \left(\frac{n-j}{n} \right)^v$$

This sum is difficult to calculate precisely but easy to calculate approximately because the first few terms dominate for large v.

For five candidates, to reach 95% probability of full coverage, we need around 12,250 voters. Six candidates need around 50,000 voters; by the time we reach ten candidates, 9.6 million voters are required to give 95% confidence that every multi-ballot turns up at least once. Note that these figures are rather conservative lower bounds: the distribution will not in fact be uniform, which will lower the probability; and in any case 95% confidence is perhaps insufficient for a critical security assumption.

These numbers are very high, and we consider them to be unrealistic. This version of the short ballot assumption is suitable only for a very small number of candidates or extremely large numbers of voters; it will not be considered further in this paper.

Assumption 2 (SBA-mini). *Every possible mini-ballot will be cast at least once.*

Under Assumption 2, we require only that each mini-ballot, rather than each multi-ballot, be cast. Clearly this is more likely to be satisfied than Assumption 1. For c candidates, there are only 2^c distinct mini-ballots, against $c \cdot 3^c$ distinct multi-ballots. For ten candidates, we therefore need coverage of only 1024 mini-ballots, rather than nearly 600,000 multi-ballots.

We will show later that this interpretation of the SBA is insufficient to prevent attacks on ThreeBallot. Since it is not a worthwhile formulation of the assumption, we need not calculate the likelihood that it will be satisfied.

Assumption 3 (SBA-mini-n). *Every possible mini-ballot will be cast at least n times (for some suitably chosen n).*

A slightly stronger interpretation in Assumption 3 requires each mini-ballot to turn up at least a certain number of times. This, of course, requires more voters than Assumption 2.

However, we will show later that this formulation is also insecure, regardless of the value of n.

1.2 Related Work

The ThreeBallot voting system has been subjected to analysis of one sort or another many times since its publication [15, 16, 17, 18, 14, 19, 13, 20, 21]. Perhaps the earliest analysis was conducted by Strauss [15, 16], who established the success probabilities of attacks for various numbers of candidates and voters with multiple races. Various attacks against the system, and in particular, reconstruction and pattern request attacks, were considered. The experiments were coded in Python, and modelled elections with a number of races on a single multi-ballot form. Clark *et al.* [17] also investigated ThreeBallot, and pointed out that the multi-ballot reveals information that can compromise voter privacy. A simulation-based analysis of the system was made by de Marneffe *et al.* [14] using the universally composable security framework [22]. Additionally, a modified system protocol in which a voter chooses her receipt before expressing her preference was proposed in [14]. This protocol was shown to guarantee election fairness, at the cost of some noise in the final tally, with the SBA assumption, and an additional assumption that most of the receipts are not known to the adversary. One drawback, however, is that the voter cannot express her preference on the mini-ballot that she has chosen as her receipt, which makes voting more complicated. Statistical results about the relation between the number of candidates in an election and the privacy level of the system were provided by Cichoń *et al.* [13] as well as a critique on the effectiveness of Strauss' attacks. Cichoń *et al.* claim that it is impossible to reconstruct voters' preferences in a single election run with two candidates with a 'reasonable number of voters'. However, the definition of weak anonymity used in [13] is much different from ours given in [11]. Considering that an individual mini-ballot can be used to construct two different multi-ballots cast for the same candidate, their definition seems necessary, but not sufficient. Hence, the observer would notice that one of the voters is not able to vote for that candidate.

A more theoretical work was carried out by Henry *et al.* [20], who focused on a two-candidates race, and determined secure ballot sizes against reconstruction and pattern requesting attacks. Finally, Küsters *et al.* [21] computationally analysed the level of privacy offered by the ThreeBallot voting system and the proposed system by de Marneffe *et al.* [14], and concluded that the latter provides better privacy than the original.

2 Modelling the ThreeBallot Voting System

In this section, we model the ThreeBallot voting system using CSP. We assume that the reader is familiar with CSP notation; for details see Roscoe's book [23].

2.1 Data-Types, Functions and Sets

We treat the multi-ballot of the ThreeBallot voting system as a board with co-ordinates. Here, a co-ordinate (i, j) defines a bubble on a mini-ballot, which is to be filled in. Thus, we have exactly three columns representing three mini-ballots, and a number of rows, which is one more than the number of candidates (the last row is allocated just for serial numbers). The size of the board is determined by these parameters: the number of voters, *VTRS*, and the number of candidates, *CNDS*. These parameters define the sets of voters, candidates and serial numbers. The data-types for voters, candidates and serial numbers are defined as $v.i$, $c.j$ and $s.k$ respectively.

We need several functions, which return a specific part of the board. For instance, $Row(i)$ returns the ith row of a multi-ballot form and $Col(j)$ is the set of bubbles on the jth column of a multi-ballot. Likewise, some functions call back the neighbouring bubbles of a given coordinate. For example, the function $adjR(i, j)$ returns the coordinates adjacent to (i, j) in the same row, similarly $adjC(i, j)$ returns the coordinates adjacent to (i, j) in the same column, and $nhdAll(i, j)$ returns all the neighbours of (i, j) in the current multi-ballot coordinates.

2.2 Processes and Channels

In this section, we define how the ThreeBallot voting system model works, and explain what information is carried on each channel. The overall system model is a parallel composition of the processes detailed below. Fig 1 illustrates the network for the ThreeBallot CSP model.

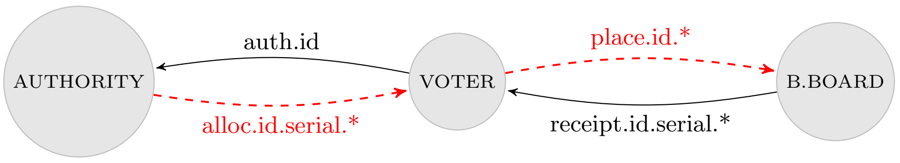


Fig. 1. ThreeBallot CSP Model Communication Channels ((- - →)private channel)

Voter Process. The voter chooses the candidate that she wants to vote for before the election is open. She then authorises herself with the election authority, and collects her multi-ballot with the *alloc* events. In the booth, the voter fills out two bubbles for the chosen candidate with the *place* events and one for the other candidates. Afterwards, she gets her receipt by choosing one of the mini-ballots allocated to her on the channel *receipt*, and leaves the booth before the election is closed.

The *VOTER*() process does *place* events in an efficient way; first a bubble from the first or second column is chosen for the candidate the voter wants to vote for then the second bubble is chosen from the other columns in a right to left fashion. Afterwards the process does one *place* event from top to bottom manner for the other candidates. The set $nhdAll(i,j) \setminus (Row(i) \cup Row(CNDS))$ is the set of bubbles left that can be filled in, and *CNDS* is the number of candidates, which also identifies the number of rows.

$$\begin{aligned}
&VOTER(id) \mathrel{\widehat=} \textstyle\bigsqcap_{c.x \in candidates} choose!id.c.x \to openElection \to auth!id \to \\
&\quad alloc.id?s1?(i1,j1) \to alloc.id?s2?(i2,j2) \to alloc.id?s3?(i3,j3) \to \\
&\qquad enterBooth!id \to \textstyle\bigsqcap_{(i,j) \in Row(x-1) \setminus Col(2)} place!id.(i,j) \to \\
&\qquad\quad \textstyle\bigsqcap_{(i1,j1) \in adjR(i,j)} place!id.(i1,j1) \to \\
&\quad VOTER'(id, nhdAll(i,j) \setminus (Row(i) \cup Row(CNDS)), \{s1,s2,s3\}, CNDS-1)
\end{aligned}$$

$$\begin{aligned}
&VOTER'(id, aSet, setsers, 0) \mathrel{\widehat=} \textstyle\bigsqcap_{rcp \in setsers} receipt.id.rcp?(i,j) \to leaveBooth!id \to \\
&\qquad closeElection \to STOP
\end{aligned}$$

$$\begin{aligned}
&VOTER'(id, aSet, setsers, cntr) \mathrel{\widehat=} place.id?(k,l) \to \\
&\quad VOTER'(id, aSet \setminus Row(k), setsers, cntr-1)
\end{aligned}$$

Thus the process representing all voters is described by the parallel composition of the voters as:

$$VOTERS \mathrel{\widehat=} \|_{id}\, VOTER(id)$$

Election Authority Process. The election official in the polling station is responsible for authenticating voters with the events *auth* and assigning the pre-printed multi-ballots (three unique serial numbers for each voter) to the voters with an *alloc* event. The authority process is defined as follows:

$$AUTHORITY \mathrel{\widehat=} openElection \to AUTHORITY'(serials)$$

$$\begin{aligned}
&AUTHORITY'(setSrls) \mathrel{\widehat=} auth?id \to \textstyle\bigsqcap_{srl \in setSrls} alloc.id.srl.(CNDS,0) \to \\
&\qquad AUTHORITY''(id, (CNDS,0), setSrls \setminus \{srl\})
\end{aligned}$$

$$\begin{aligned}
&AUTHORITY''(id, coord, \emptyset) \mathrel{\widehat=} closeElection \to STOP \\
&AUTHORITY''(id, (CNDS,2), setSerials) \mathrel{\widehat=} AUTHORITY'(setSerials) \\
&AUTHORITY''(id, (CNDS,i), setSerials) \mathrel{\widehat=} \\
&\qquad \textstyle\bigsqcap_{srl \in setSerials} alloc.id.srl.(CNDS,i+1) \to \\
&\qquad AUTHORITY''(id, (CNDS,i+1), setSerials \setminus \{srl\})
\end{aligned}$$

The authority opens the election, authorizes the voters, and assigns serial numbers to each mini-ballot with the *alloc* events. After the election, the authority performs *closeElection*, after which no more ballots can be allocated.

The Bulletin Board Process. The process *B_BOARD* operates as a bulletin board where the cast mini-ballots are published. The votes are collected while the voters cast their mini-ballots. Thus, the process keeps a record of the serial numbers and the bubbles that are filled in the set *Bag*. The mini-ballots are published with the *pub* event after the election is closed.

$$BOARD(srl) \mathrel{\widehat{=}} alloc?id!srl?(i,j) \rightarrow BOARD'(\emptyset, srl, (i,j))$$

$$\begin{aligned} BOARD'(Bag, srl, (i,j)) \mathrel{\widehat{=}} & \; place.id?(m,n) : Col(j) \rightarrow BOARD'(Bag \cup \{m\}, srl, (i,j)) \\ & \Box \; receipt?id!srl.Bag \rightarrow BOARD''(srl, Bag) \\ & \Box \; BOARD''(srl, Bag) \end{aligned}$$

$$BOARD''(srl, Bag) \mathrel{\widehat{=}} closeElection \rightarrow pub.srl.Bag \rightarrow bagempty \rightarrow STOP$$

$$B\_BOARD \mathrel{\widehat{=}} openElection \rightarrow \|_{serials} BOARD(serials)$$

Counter Process. The other important system process is *COUNTERS*. This works as an election authority, which counts the votes that are published on the bulletin board. The process keeps record of *place* events for each candidate. When all of the *place* events have occurred, it performs a *bagempty* event on which all *COUNTERS* processes synchronise. With the *total* event the number of total votes for each candidate is published.

$$\begin{aligned} COUNTER(cand, r) \mathrel{\widehat{=}} & \; place?id?(i,j) \rightarrow COUNTER(cand, r+1) \\ & \Box \; bagempty \rightarrow total!cand!r \rightarrow STOP \end{aligned}$$

$$COUNTERS \mathrel{\widehat{=}} \|_{candidates} COUNTER(cand, 0)$$

System Process. The ThreeBallot voting system model is the parallel composition of the processes defined previously. Hence, the composition is defined as follows:

$$SYSTEM \mathrel{\widehat{=}} VOTERS \parallel AUTHORITY \parallel BOOTH \parallel B\_BOARD \parallel COUNTERS$$

3 Automated Anonymity Verification

Our analysis of ThreeBallot uses the formal anonymity definition given in [11]. The definition of anonymity for the voting systems, also called weak anonymity, is based on observational equivalence and expressed as follows:

Definition 1. *The process P is weakly anonymous on a set of channels C of type T if:*

$$P[\![{}^{c.x,\, d.x}/_{d.x,\, c.x} \mid x \in T]\!] \equiv_{\mathrm{T}} P \tag{1}$$

for any $c, d \in C$

That is, when the two channels $c.x$ and $d.x$ are swapped over for all values of x, if the resulting process is indistinguishable from the original process, P, from an observer's point of view, then the process provides anonymity.

It is over channel *choose* that the voter determines a choice of candidate; consequently, the channels that need to be swapped over are: $choose.v.1.c.x$ and $choose.v.2.c.x$ for $c.x \in candidates$. Therefore, the anonymity specification for ThreeBallot CSP model ($SYSTEM$) is checked by the trace equivalence:

$$SYSTEM[\![^{choose.v.1.c.x,\, choose.v.2.c.x}/_{choose.v.2.c.x,\, choose.v.1.c.x}]\!] \equiv_T SYSTEM$$

As the anonymity property of the system is checked from an observer's point of view, the observer's inability to see sensitive information is extremely important. He is able to see all the public channels, but not the private channels: *alloc* and *place*. Therefore, these private channels need to be hidden.

$$ABS\_SYS \mathrel{\widehat{=}} SYSTEM \setminus \{|\, alloc, place \,|\}$$

As can be seen above, the normal system is ABS_SYS, and the system where we swap two votes is $SPEC$. Therefore, if the two systems are observationally equivalent then the system provides anonymity.

$$SPEC \mathrel{\widehat{=}} ABS\_SYS[\![^{choose.v.1.c.x,\, choose.v.2.c.x}/_{choose.v.2.c.x,\, choose.v.1.c.x}]\!]$$

We assume that the adversary in our model is able to see all *receipt* events; i.e., he can see all the receipts taken in an election. (This is a strong assumption; however, if the system is secure under this assumption, it will also be secure with an adversary who sees only some receipts.)

3.1 Results for the ThreeBallot Model with No SBA

Unsurprisingly, the refinement $SPEC \equiv_T ABS\_SYS$ does not hold for our ThreeBallot voting system model. This is because there are situations in which a reconstruction attack is possible: that is, a coercer who has seen receipts for v_1 and v_2 can infer that they voted respectively for c_1 and c_2 because there is no way of constructing a complete set of valid multi-ballots in which v_1 and v_2 vote for c_2 and c_1 respectively. Whether the election run provides anonymity entirely depends on how the voters fill their multi-ballots, and also on which mini-ballots they choose as receipts.

The following counter-examples from different voting scenarios give useful intuition about in what situations anonymity is not satisfied.

Examples of Privacy Violations of ThreeBallot

Example 1. The first counter-example is taken from a protocol run with two voters, v_1 and v_2, and two candidates, c_1 and c_2. The FDR2 model checker returns several counter-examples which violate anonymity. We examine one of these traces here, illustrating the receipts taken by the voters and the mini-ballots displayed on the bulletin board. The following illustrated examples are the election runs from the observer's point of view.

The counter-example trace shows that in a voting scenario as in Table 2, where v_1 chooses to vote for c_1, and v_2 votes for c_2, if the voters take s_2 and s_3 respectively as their receipts, the observer is able to reconstruct the multi-ballots from the public mini-ballots on the bulletin board. There is no possible reconstruction where the votes were cast the other way round. Therefore, the observer is able to say who voted for whom in this ThreeBallot election run.

Table 2. Voting scenario 1

Receipts		**Mini-ballots on BB**			
●	○	○	●	●	○
●	●	○	○	●	○
s_2	s_3	s_0	s_1	s_4	s_5

Table 3. Reconstruction attack 1

choose.v.1.c.1			**choose.v.2.c.2**		
●	●	○	○	○	●
●	○	○	●	○	●
s_2	s_1	s_5	s_3	s_0	s_4

With the public information shown on the bulletin board and the receipts that the voters share with the coercer, the only way of reconstructing these votes is illustrated in Table 3. The mini-ballots s_0 and s_5 can be swapped. However, it does not affect the way the voters have voted.

Example 2. In an election with three voters and two candidates, as depicted in Table 4, when voter v_1 votes for c_1, voter v_2 votes for c_2, and voter v_3 votes for c_1, with the receipts s_1, s_2 and s_0 respectively, voter v_1 can be seen not to have voted for c_2. Table 5 shows the only possible reconstruction.

Table 4. Example 2. voting scenario

Receipts			**Mini-ballots on the BB**					
●	○	○	●	○	○	●	●	●
○	●	●	●	○	●	○	○	○
s_1	s_2	s_0	s_3	s_4	s_5	s_6	s_7	s_8

Table 5. Example 2. reconstruction attack

choose.v.1.c.1			**choose.v.2.c.2**			**choose.v.3.c.1**		
●	●	○	○	○	●	○	●	●
○	●	○	●	●	○	●	○	○
s_1	s_3	s_4	s_2	s_5	s_6	s_0	s_7	s_8

3.2 Short Ballot Assumption

We now analyse the ThreeBallot voting system under two of the three possible interpretations of the SBA that were given earlier: Assumptions 2 and 3. (Recall that Assumption 1 seemed implausible unless there were only very few candidates.)

Analysis Under the SBA-Mini. Suppose we adopt Assumption 2, under all possible mini-ballots are assumed to appear on the bulletin board at least once at the end of the election. We give here a simple counter-example to show that ThreeBallot does not provide anonymity. In the example in Table 6, receipt s_0 has two possible completions: it could be combined with s_2 and s_4 or s_8 (as depicted in Table 7), or with s_5 and s_7. But in either case, it represents a vote for the third candidate.

Table 6. An example voting scenario: all possible mini-ballots appear on the bulletin board

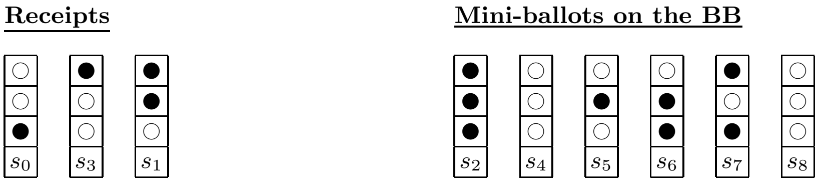
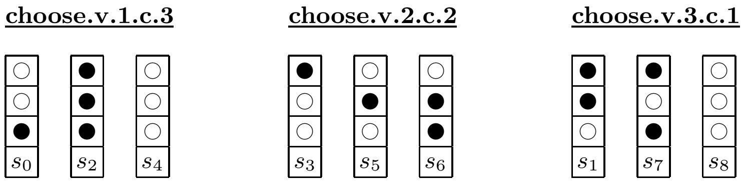


Table 7. Reconstruction attack



Analysis Under SBA-Mini-n. Suppose now that we adopt Assumption 3, which ensures that every possible mini-ballot will appear on the bulletin board at least n times for some suitable value of n. We show here that this is insufficient regardless of the value of n.

We start by observing that a fully filled mini-ballot can be combined only with an empty mini-ballot and a singleton. Additionally, any possible mini-ballot m that is not empty, fully filled or a singleton can be turned into a completed multi-ballot that does not contain a fully filled mini-ballot or a singleton. This can be done by combining it with another mini-ballot that is the complement of m but with one extra bubble, and an empty mini-ballot.

We can reach a bulletin board that displays at least n copies of every possible mini-ballot in the following way. For each possible mini-ballot that is not empty, fully filled or a singleton, we turn it into a multi-ballot as described above, and add it to the board. This gives us at least n copies of everything except singletons and fully filled mini-ballots.

Now each possible singleton should be combined with a fully filled mini-ballot and an empty mini-ballot. We add n copies of each such multi-ballot to the board. This means that every possible mini-ballot now appears at least n times.

However, any voter taking a singleton as a receipt will have no anonymity. The number of fully filled mini-ballots is the same as the number of singletons; and since each fully filled ballot must be combined with a singleton and a blank, it follows that the voter's receipt must have been part of such a multi-ballot. But in that case the mini-ballot reveals the candidate that the voter selected.

Hence no value of n is sufficient to guarantee anonymity in ThreeBallot.

SBA-Pro: A Better Formulation. We have seen that the interpretations of the SBA given so far are either not enough or unrealistic. We now give a much more plausible short ballot assumption that is demonstrably strong enough for ThreeBallot.

Assumption 4 (SBA-pro). *Let M be the set of all mini-ballots cast during the election; $R \subset M$ is the set of all receipts that are known to the adversary. We introduce a partial function vote such that $vote(m_1, m_2, m_3) = c$ whenever the three mini-ballots m_1, m_2 and m_3 together form a valid multi-ballot that represents a vote for c. Additionally, for any two mini-ballots m_1 and m_2, we say that $m_1 \sim m_2$ if and only if they contain the same sequence of vote marks (that is, $m_1 = m_2$ except possibly for the serial numbers).*

For every $r \in R$ and every candidate c, there was a vote cast consisting of three (unordered) mini-ballots m_1, m_2, m_3 such that

1. *$r \sim m_1$;*
2. *$vote(m_1, m_2, m_3) = c$;*
3. *$m_2, m_3 \in M \setminus R$.*

Informally, this interpretation says that for every receipt known to the adversary, there was an equivalent one used in a multi-ballot for each of the candidates in the election.

Theorem 1. *Assumption 4 is strong enough to prevent reconstruction attacks in ThreeBallot.*

Proof. The key to the proof is the observation that if $m \sim m'$ then we must have $vote(m, m_2, m_3) = vote(m', m_2, m_3)$. This is clear from the fact that m and m' can differ only in serial number, and the serial numbers are not relevant for determining which candidate received the vote cast by a multi-ballot.

Suppose that $r \in R$, and the adversary wishes to determine which candidate received the vote cast that included r. We can see that any candidate is possible. Suppose that r did in fact occur in a multi-ballot along with m_1 and m_2, as a vote for c. For any other candidate c', there was a multi-ballot cast containing

m_3, m_4, m_5 such that $vote(m_3, m_4, m_5) = c'$ and $r \sim m_3$, and with m_4 and m_5 not known to the adversary.

But this means that the adversary cannot distinguish the following two possibilities:

1. a ballot of (r, m_1, m_2) for c, and a ballot of (m_3, m_4, m_5) for c';
2. a ballot of (m_3, m_1, m_2) for c, and a ballot of (r, m_4, m_5) for c'.

In each case, the set of mini-ballots used by this partial reconstruction is the same, so it cannot affect further reconstruction of the remaining mini-ballots. In one case, r was used to vote for c, and in another case, for c'; and since c' was arbitrarily chosen, we conclude that r could equally have been used to vote for any candidate.

To see the improved plausibility of this interpretation, suppose the adversary has knowledge of r receipts in an election run with n candidates. The SBA-pro requires at least $n \cdot r$ multi-ballots of the right type to have been cast to protect anonymity. By contrast, the SBA-multi requires at least $n \cdot 3^n$ other appropriate multi-ballots. As long as r is small, the SBA-pro is much less demanding compared with the SBA-multi. For instance, in an election with 10 candidates, the SBA-multi needs at least 590,490 multi-ballots. Unless the adversary has seen somewhere in the order of 59,000 receipts, the SBA-pro is much more likely to be satisfied.

This efficiency argument is not absolute: to formalise it would require a full voter model; that is, it would need a probability distribution over multi-ballots cast in the election. Producing such a model is probably unrealistic, since it would be affected by the prevailing political landscape at the time of the election; it is in any case outside the scope of this paper.

3.3 Verified Privacy Cases

Apart from the short-ballot assumption, several slight modifications of ThreeBallot have been proposed to help the system provide absolute anonymity. Using FDR we were able to verify these modified systems against reconstruction attacks. We have automatically verified a ThreeBallot model that allows voters to exchange their receipts; and we analyse the system with an additional constraint that voters must fill in at least one bubble in every column.

Floating/Exchanging Receipts. Rivest [5] suggests a possible improvement to the original ThreeBallot scheme with the idea of exchanging receipts in the polling station. Each voter puts her receipt in a box, and takes someone else's receipt. Indeed, this idea can be used in any paper-based election system. If we let voters take a random receipt from the box in the polling station, then this eliminates reconstruction attacks as well as pattern-matching (Italian) attacks because the adversary does not have any knowledge of any part of the voter's ballot. Although the adversary may be able to reconstruct valid multi-ballots, he cannot link them to voters. We have verified using FDR that the modified scheme, where the voters are allowed to exchange their receipts.

No Single Mini-Ballot Left Blank. We here add a condition that voters must fill out at least one bubble on each mini-ballot. For the two candidate case, there are only two ways of filling a mini-ballot, and thus only two different receipt that can be taken by voters. We have modified our model to provide automatic verification that this condition is sufficient to guarantee anonymity with two candidates. However, in an election where there are more candidates than two, although intuitively the system provides better probabilistic anonymity than the original, it cannot guarantee voter anonymity.

4 Conclusion

In this paper, we have demonstrated that the ThreeBallot voting system is vulnerable to privacy-related attacks, especially reconstruction attacks, even under some plausible interpretations of the short ballot assumption.

In our analysis, we have used an abstracted CSP model of ThreeBallot, which is defined as the parallel composition of agents in the system. We model the adversary in the analysis as an outsider/observer, who can see all the public channels, including what each voter takes as a receipt. We have given a number of examples for different voting scenarios, demonstrating that ThreeBallot does not provide anonymity under various formulations of the short ballot assumption. We have in addition given a reasonable and plausible interpretation of the short ballot assumption that does in fact prevent reconstruction attacks.

Finally, we have considered two different versions of ThreeBallot that we were able to analyse automatically using FDR; namely, exchanging receipts and no single mini-ballot left blank.

Because of the state space limitation that all model checking tools suffer from, we were able to analyse the models with a limited number of agents. In most cases, the restriction did not affect the analysis of the systems and assumptions; however, as the short-ballot assumptions require a large number of mini-ballots, we were not able to demonstrate automatic verification in such cases; however, we have supplied hand proofs where appropriate. Table 8 illustrates the ThreeBallot verification times ("–" means no result is produced in a reasonable time).

Table 8. FDR verification times for ThreeBallot versions

	Original		**No mini-ballot empty**		**All mini-ballots appear**	
	States	Time	States	Time	States	Time
2 vtrs 2 cnds	239, 905	$7.8'$	56, 841	$5.3'$	240, 055	$7.0'$
2 vtrs 3 cnds	4, 139, 347	$1''41.8'$	1, 435, 926	$38.3'$	4, 165, 428	$1''40.1'$
3 vtrs 2 cnds	–	–	67, 409, 391	$22''49.3'$	–	–

References

[1] Chaum, D.: Untraceable electronic mail, return addresses, and digital pseu-donyms. Communications of the ACM 24, 84–90 (1981)

[2] Fujioka, A., Okamoto, T., Ohta, K.: A practical secret voting scheme for large scale elections. In: Zheng, Y., Seberry, J. (eds.) AUSCRYPT 1992. LNCS, vol. 718, pp. 244–251. Springer, Heidelberg (1993)
[3] Juels, A., Catalano, D., Jakobsson, M.: Coercion-resistant electronic elections. IACR Cryptology ePrint Archive 2002, 165 (2002)
[4] Chaum, D., Ryan, P.Y.A., Schneider, S.: A practical voter-verifiable election scheme. In: De Capitani di Vimercati, S., Syverson, P.F., Gollmann, D. (eds.) ESORICS 2005. LNCS, vol. 3679, pp. 118–139. Springer, Heidelberg (2005)
[5] Rivest, R.L.: The ThreeBallot voting system (2006)
[6] Hoare, C.A.R.: Communicating Sequential Processes. Communications of the ACM 21, 666–677 (1978)
[7] Gardiner, P., Goldsmith, M., Hulance, J., Jackson, D., Roscoe, B., Scattergood, B., Armstrong, B.: FDR2 user manual
[8] Backes, M., Hritcu, C., Maffei, M.: Automated verification of remote electronic voting protocols in the applied pi-calculus. In: CSF, pp. 195–209 (2008)
[9] Smyth, B.: Formal verification of cryptographic protocols with automated reasoning. PhD thesis, School of Computer Science, University of Birmingham (2011)
[10] Ryan, P.Y.A., Schneider, S.A.: Prêt à Voter with re-encryption mixes. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 313–326. Springer, Heidelberg (2006)
[11] Moran, M., Heather, J., Schneider, S.: Verifying anonymity in voting systems using CSP. Formal Aspects of Computing, 1–36 (2012)
[12] Rivest, R.L., Smith, W.D.: Three voting protocols: ThreeBallot, VAV, and Twin. In: Proceedings of USENIX/ACCURATE Electronic Voting Technology (EVT). Press (2007)
[13] Cichoń, J., Kutyłowski, M., Węglorz, B.: Short ballot assumption and threeballot voting protocol. In: Geffert, V., Karhumäki, J., Bertoni, A., Preneel, B., Návrat, P., Bieliková, M. (eds.) SOFSEM 2008. LNCS, vol. 4910, pp. 585–598. Springer, Heidelberg (2008)
[14] de Marneffe, O., Pereira, O., Quisquater, J.-J.: Simulation-based analysis of E2E voting systems. In: Alkassar, A., Volkamer, M. (eds.) VOTE-ID 2007. LNCS, vol. 4896, pp. 137–149. Springer, Heidelberg (2007)
[15] Strauss, C.: The trouble with triples: A critical review of the triple ballot (3ballot) scheme part1 (2006)
[16] Strauss, C.: A critical review of the triple ballot voting system, part2: Crack- ing the triple ballot encryption (2006)
[17] Clark, J., Essex, A., Adams, C.: On the security of ballot receipts in E2E voting systems. In: IAVoSS Workshop On Trustworthy Elections (WOTE) (July 2007)
[18] Appel, A.W.: How to defeat Rivest's ThreeBallot voting system (2007)
[19] Tjøstheim, T., Peacock, T., Ryan, P.Y.A.: A case study in system-based analysis: The ThreeBallot voting system and Prêt à Voter. In: VoComp (2007)
[20] Henry, K., Stinson, D.R., Sui, J.: The effectiveness of receipt-based attacks on ThreeBallot. Trans. Info. For. Sec. 4(4), 699–707 (2009)
[21] Küsters, R., Truderung, T., Vogt, A.: Verifiability, privacy, and coercion-resistance: New insights from a case study. In: 2011 IEEE Symposium on Security and Privacy (SP), pp. 538–553 (May 2011)
[22] Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: Proc. 42nd IEEE Symp. Foundations of Computer Science, pp. 136–145 (2001)
[23] Roscoe, A.W.: Understanding Concurrent Systems, 1st edn. Springer-Verlag New York, Inc., New York (2010)

Compositional Verification of Software Product Lines

Jean-Vivien Millo[1,2], S. Ramesh[2], Shankara Narayanan Krishna[3], and Ganesh Khandu Narwane[4]

[1] EPI AOSTE, INRIA Sophia-Antipolis, France
[2] Global General Motors R&D, TCI Bangalore, India
[3] Department of CSE, IIT Bombay, Mumbai, India
[4] Homi Bhabha National Institute, Mumbai, India

Abstract. This paper presents a novel approach to the design verification of Software Product Lines (SPL). The proposed approach assumes that the requirements and designs at the feature level are modeled as finite state machines with variability information. The variability information at the requirement and design levels are expressed differently and at different levels of abstraction. Also the proposed approach supports verification of SPL in which new features and variability may be added incrementally. Given the design and requirements of an SPL, the proposed design verification method ensures that every product at the design level behaviourally conforms to a product at the requirement level. The conformance procedure is compositional in the sense that the verification of an entire SPL consisting of multiple features is reduced to the verification of the individual features. The method has been implemented and demonstrated in a prototype tool SPLEnD (SPL Engine for Design Verification) on a couple of fairly large case studies.

1 Introduction

Large industrial software systems are often developed as *Software Product Line* (SPL) with a common core set of features which are developed once and reused across all the products. The products in an SPL differ on a small set of features which are specified using *variation points*. The focus of this paper is on modeling and analysis of SPLs which have drawn the attention of researchers recently [1,2,3]. Many approaches have been proposed to describe SPLs, the most prominent one being *feature diagrams*. These approaches seem to assume a global view of SPL as they start with a complete list of features and the variation points using a single vocabulary. All the subsequent SPL assets, like requirement documents, design models, source codes, test cases, documentations, share the same definition and vocabulary [4,5]. However, the assumption of a single homogeneous and global view of variability description is inapplicable in many practical settings, where there is no top level complete description of features and variabilities. They often evolve during the long lifetime of an SPL as new features and variabilities are added during the evolution. Further, SPL developers tend to use different representations and vocabulary of variability at different stages of development: at the requirement level, a more abstract and intuitive description of variation points are used, while at the design level, the efficiency of implementation of variation points is of primary concern. For example, consider the case of an automotive SPL, where one variation point is the region of sales (eg. Asia Pacific, Europe, North America etc). At the requirement level, this variation point is expressed directly as an enumeration variable assuming one value for every region. Whereas, at the design level, the variation point is expressed using two or three boolean variables; by setting the values of the boolean variable appropriately, the behaviour specific to a region is selected at the time of deployment.

We present a design verification approach that is more suited to the above kind of evolving SPLs in which different representation of variabilities would be used at the requirement and design level. One natural and unique problem that arises in this context is to relate formally the variation points expressed at different levels of abstractions. Another challenge is the analysis complexity: the number of products is exponential in the number of variation points and hence product centric analyses are not scalable. We propose a compositional approach in which every feature of the SPL is first analyzed independently; the per-feature analysis results are then combined to get the analysis result for the whole SPL. For capturing variability in the behaviour of an SPL, we have extended the standard finite state machine model, which we call *Finite State Machines with Variability*, in short, *FSMv*. The behaviour and variability of a feature at the requirement and design level can be modeled using FSMv. We define a conformance relation between FSMvs to relate the requirement and design models. This relation is based upon the standard language containment of state machines. One unique feature of FSMv is that it provides a compositional operator for composing the feature state machines to obtain a model for an SPL. This operator thus enables incremental addition of features and variabilities. The proposed verification approach exploits the compositional structure of the SPL models to contain the analysis complexity.

Figure 1 summarizes the proposed approach. It shows an SPL composed of features f_1 to f_n. Each feature has an FSMv model of its requirements (called

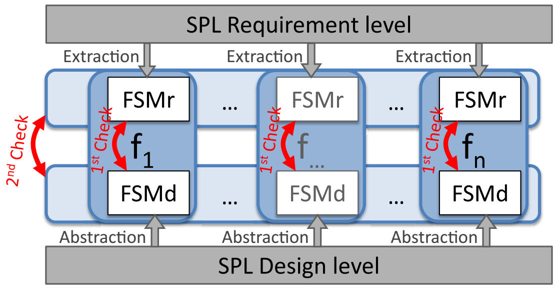


Fig. 1. The proposed verification approach

FSMr) and an FSMv model derived from its design (called FSMd). The proposed analysis method checks whether the FSMd of every feature conforms to its FSMr (1^{st} check). The output of this first step is a conformance relation Φ_i between each pair of $FSMr_i$ and $FSMd_i$. The obtained conformance relations $\Phi_1, \ldots, \Phi_n$ are then used to check whether the actual behaviour of the entire SPL conforms to the expected one (2^{nd} check). The 2^{nd} check is done by synthesizing a Quantified Boolean Formula (QBF) and answering its satisfiability. There is no need to build the entire behavioural model of the SPL in the second step. We have built a prototype tool SPLEnD based upon this approach. This tool performs the first check using SPIN [6] while the well-known QBF SAT solver CirQit [7] is used for the second step. We have experimented with the tool using modest industrial size examples with very encouraging results.

1.1 Related Work

In this section, we survey related work in five broad themes : *Feature Based Analysis*, *Behavioural Conformance*, *Compositional Verification*, *SAT Solving* and *SPL Tools.*

Feature Based Analysis: [8] explores feature-aware verification to automatically detect feature interactions in a software product line. For this, a language is developed to specify individual features in separate and composable units; based on these feature-local specifications, feature interactions are detected in a product line by either (i) generating all the products and checking them one by one, or (ii) by generating one product that contains all the features. The *email* product line with 10 features and 40 products, with 27 feature interactions was checked. A programming language oriented approach is presented in [9], where, a core calculus for feature composition is developed. The features may contain various kinds of software artefacts, like source code in various languages, models and documents. The composition is done uniformly across features with different artifacts in a type-safe way. In a third approach, Fisler et al [10] view features as state machines, and CTL model checking is used to verify properties of individual features. Compositional verification of features is done by checking the consistency of interface labels assigned by the CTL model checking algorithm at the feature level.

Behavioural Conformance: [11] proposes the use of modal transition systems (MTS) over labelled transition systems for modelling and analysis of product line architectural behaviour. MTS can model optional and required behaviour via *may* and *must* transitions. A conformance algorithm for MTS is then presented: a fixed point algorithm that computes cartesian product of states, and eliminates pairs that are invalid according to the relation. A second line of work is FTS^+, proposed in [2] that has some similarities with FSMv, but has a motivational difference. The aim of FTS^+ is to model the entire SPL and hence there is a single global machine with a single global vocabulary for expressing variabilities; the variability information represents the presence/absence of features in the SPL. In contrast, our approach is based upon a different view of

SPL: a feature with variability is an increment in functionality and an SPL is a collection of features. We use a single FSMv to model a feature and a whole SPL is modeled as a parallel composition of FSMv machines. The difference in viewpoint has another consequence: FTS$^+$ models, since they model the entire SPL, tend to be large and hence has high analysis complexity; some abstraction techniques are hence used [3]. Whereas, each FSMv models a fraction of functionality and hence can be analysed easily. Further, the entire SPL can be modeled as composition of FSMvs and can be efficiently analysed using composition techniques. In a third approach, [12] uses MTS for modelling product behaviour and use the logic MHML for model checking. The approaches in [12] as well as [2] use transition systems for expressing system behaviour; feature variability constraints are expressed using feature diagrams in [2], while in [12], MHML is used to do this. [2] needs an extra component, a logic for checking properties, while in the case of [12], the MTS+MHML framework is sufficient.

Compositional Verification: [13] proposes compositional verification for hierachical SPLs. Here, Simple Hierarchical Variability Models (SHVM) are used to specify the variability of product artifacts. However, in an SHVM, the number of derivable products is restricted by the fact that there is no means of defining constraints between variation points. On an experimetal setup, [14] uses Event-B composition techniques for feature based product line development. A feature is considered as a basic modular unit in the Rodin tool, and two case studies have been evaluated.

SAT Solving: [15] was the first to propose the use of propositional logic for expressing relationships between requirements in a product line model. Using this, a product line model can be represented as a logical expression; this can be instantiated by the selected requirements. Further, it can be chcked if the selected set is valid or not. Further, [16] explores the fundamental connection between feature diagrams, grammars and propositional logic formulae. This connection paved the way for the use of SAT solvers that provide automated support to debug feature models.

Other Approaches and SPL Tools: Many other behavioural models have also been proposed [17,18,19,20] which are usually coupled with a variability model such as OVM [5], the Czarnecki feature model [4], or VPM [21] to attain a fair level of variability expressibility. Unlike all these approaches, FSMv capture the variability in an explicit way which we find more intuitive. The Variation Point Model (VPM) of Hassan Gomaa [21] distinguishes between variability at the requirement and design levels but no design verification approach has been presented. In a recent paper, Jorges *et al.* [22] present a constraint based approach for variability modeling. Here, architectural as well as behavioural constraints are captured by using temporal logics; synthesis algorithms are then used to compute solutions. Kathrin Berg *et al.* [23] proposes a model for variability handling throughout the life cycle of the SPL. Andreas Metzger *et al* [24] and Riebisch M. *et al* [25] provide a similar approach but they do not consider the behavioural aspect. In our proposed approach, we extract the relation between requirement and design level variability from a behavioural analysis. [26] present a tool VMC, for the modeling and analysis of product lines. The product family is represented as an MTS, along with extra variability constraints, and all the valid products are automatically generated. The tool implements the algorithm presented in [12]. A demonstration of the main features of VMC can be seen in [27]. Kathi Fisler *et al* [28] have developed an analysis based on three-valued model checking of automata defined using step-wise refinement. Later on, Jing Liu *et al.* [29] have revisited Fisler's approach to provide a much more efficient method. Recently, Maxime Cordy *et al.* have extended Fisler's approach to LTL formula [30]. Kim Lauenroth *et al.* [31] as well as Andreas Classen *et al.* [2,3], and Gruler *et al.* [32] have developed model checking methods for SPL behaviour. These methods are based on the verification of LTL/CTL/modal μ calculus formula.

All these verification methods assume a global view of variability and hence the representation of variability information is identical in both specification and the design. By contrast, in our work the specification and design involve variability information at different levels of abstraction and hence one needs mapping information between the two levels. Furthermore, our formalism allows incremental addition of functionality and variability and enables compositional verification.

2 Design Verification of a Single Feature

An SPL, in general consists of multiple features, each feature having different functionality and variability. A typical body control software of an automotive system is an SPL that has several features such as door lock, lighting, seat control etc. Each of these features has a distinct function and variability. For example, the locking behaviour of a door lock function has a variation point called *transmission type*. If the transmission type is manual then the door is locked after the speed of the vehicle exceeds a certain threshold value; for automatic transmission, the door is locked when the gear position is shifted out of park. In this section we will focus on modeling and relating the design of a *single feature* to its requirement.

2.1 FSMv and Language Refinement

Finite State Machines with Variability (FSMv) is an extension of finite state machines, to represent all possible behaviours of a feature. Let Var be a finite set of variables, each taking a value ranging over a finite set of values. Let $x \in Var$, and let $Dom(x)$ denote the set of values x can assume. The set of atomic formulae we consider are $x = a$, $x \neq a$, for $a \in Dom(x)$, and $x = y$, $x \neq y$ for $x, y \in Var$. Let A_{Var} denote the set of atomic formulae over Var. Let α represent a typical element of A_{Var}. Define $\Delta ::= \alpha \mid \neg\Delta \mid \Delta \wedge \Delta \mid \Delta \vee \Delta \mid \Delta \Rightarrow \Delta$ to be the set of all well formed predicates over Var.

Definition 1 (FSMv). *An FSMv is a tuple $\mathcal{A} = \langle Q, q_0, \Sigma, Var, E, \rho \rangle$ where: (1) Q is a finite set of states; q_0 is the initial state; (2) Σ is a finite set of events; (3) Var is a finite set of variables; (4) $E \subseteq Q \times \Delta \times \Sigma \times Q$ gives the* *set of transitions. A transition $t = (s, g, a, s')$ represents a transition from state s to state s' on event a; the predicate g is called a guard of the transition t; g is consistent and defines the variability domain of the transition; (5) $\rho \in \Delta$ is a consistent predicate called the global predicate.*

The variables in Var determine the variability allowed in the feature with each possible valuation of the variables corresponding to a variant. The allowed values of the variables are constrained by the global predicate ρ. For example, if ρ is $((x = 1) \vee (x = 2)) \wedge (x = y - 1)$, then the allowed variants are those for which the values for the pairs (x, y) are $(1, 2), (2, 3)$. The predicate in a transition determines the variants to which the transition is applicable. While drawing a transition $t = (s, g, a, s')$, the edge connecting s to s' is decorated with $g : a$. When g is true, we simply write a on the edge.

Definition 2 (Configuration). *A configuration, denoted by π, is an assignment of values to the variables in Var. The set of all configurations is denoted by Π_{Var}, or Π, when Var is clear from the context. Define $\Pi(\rho) = \{\pi \mid \pi \models \rho\}$ to be the set of all those configurations that satisfy ρ. The elements of $\Pi(\rho)$ are called valid configurations. Given a valid configuration π and a transition $t = (s, g, a, s')$, we say that t is enabled by π if $\pi \models g$.*

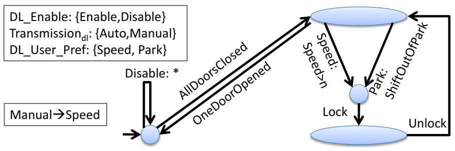


Fig. 2. The FSMv of the feature *Door lock*

As a concrete example of an FSMv, consider the feature *Door lock* in automotive SPL which controls the locking of the doors when the vehicle starts. The expected behaviour of this feature is modeled using the FSMv Req_{dl} described pictorially in Figure 2. In the initial state, this feature becomes active when all the doors are closed. The doors are locked when either the speed of the vehicle exceeds a predefined value or the gear is shifted out of park. An unlock event reactivates the feature. There are four configurations for this feature all of which are described using the three variables: DL_Enable, $Transmission_{dl}$ and DL_User_Pref. The top box denotes the values that these variables can assume, and the bottom box gives the global predicate (ρ) associated with the machine. ρ ensures that in every valid configuration, the variable $Transmission_{dl}$ having the value $Manual$ implies that DL_User_Pref takes the value *Speed*. This captures the fact that in manual transmission, there is no park position on the gearbox. To avoid clutter, we have replaced guards of the form $x = i$ with i in the figure. So, the self loop *Disable* : $*$ stands for *DL_Enable* = *Disable* : $*$. It means that when *DL_Enable* assumes the value *Disable*, it stalls on any event.

Requirement against Design. In the requirement of a product line, the variability is usually discussed in terms of variation points, which are at a high level of abstraction and focused on clarity and expressibility. The restriction of the possible configurations is expressed as general constraints on these variation points, e.g., the global predicate $Manual \implies Speed$ in the *Door lock* example. In contrast, in a design, the variability description is constrained by efficiency, implementability, ease of reconfiguration and deployment considerations. For instance, in the automotive applications, one often finds *calibration parameters* ranging over a set of boolean values. Further, the constraint on the calibration parameters (ρ) takes the special form of the list of the possible configurations of the calibration parameters in order to easily configure the design.

FSMv can capture both the design as well as the requirements of a feature. We distinguish the requirement and design models by denoting them FSMr and FSMd respectively. Figure 2 presents the FSMr, Req_{dl}, of the feature *Door lock*. The FSMd, Des_{dl}, of the feature *Door lock* is presented in Figure 3. The structure of Des_{dl} is similar to Req_{dl} except that the top elliptical shaped state in Figure 2 is split into two states (the top and the bottom elliptical shaped states) in Figure 3. The top state is for auto-transmission whereas the bottom one is for manual transmission as can be seen from the configuration label of the two transitions going from the initial state. Two variables $Cp1$ and $Cp2$ encode the possible configurations in the FSMd. The box in Figure 3 depicts the set of possible values of these. $Cp1 = Auto$ corresponds to the configuration in which the transmission is *Auto* whereas $Cp1 = Moff$ corresponds to either the manual transmission or the case when $Cp1$ is disabled; similarly, $Cp2 = Speed$ means that the user preference is set on *Speed*, while $Cp2 = Poff$ means either *Park* or the case when $Cp2$ is disabled.

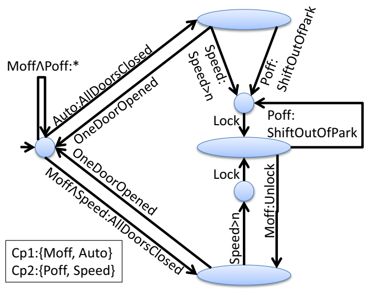


Fig. 3. Des_{dl}: the FSMd abstracted from the design of the feature *Door lock*

2.2 Variants of FSMv and Conformance

Having described the design and requirement behaviour of a feature f using FSMd and FSMr respectively, we now define the notions of variants and conformance. A variant of an FSMv corresponds to one of the several possible behaviours of the feature (at the design, requirement level respectively). Given a feature f, and a (FSMd, FSMr) pair corresponding to f, we say that the design of f conforms to the requirements of f, iff every variant of the FSMd has a corresponding FSMr variant.

Definition 3 (Variant of an FSMv). *Let $\mathcal{A} = \langle Q, q_0, \Sigma, Var, E, \rho\rangle$ be an FSMv and $\pi \in \Pi(\rho)$ be a valid configuration of $\mathcal{A}$. A variant of $\mathcal{A}$ is an FSM obtained by retaining only transitions $t = (s, g, a, s')$, and states s, s' such that $\pi \models g$. Once the relevant states and transitions are identified, we remove the guards g from all the transitions; ρ is also removed. The resultant FSM is denoted $\mathcal{A} \downarrow \pi$.*

In the example of FSMr for the feature *Door lock*, the variant $Req_{dl} \downarrow \langle Enable$, $Auto$, $Park\rangle$ does not contain the transitions with the event $Speed > n$ and $*$. We compare the FSMd and FSMr of a feature f using their variants. Given an FSMv $\mathcal{A}$, we associate with each configuration π of $\mathcal{A}$ the language of the FSM $\mathcal{A} \downarrow \pi$, denoted by $L(\mathcal{A} \downarrow \pi)$. We say that an FSMd $\mathcal{A}_d$ conforms to an FSMr $\mathcal{A}_r$ if and only if the behaviour of every variant of $\mathcal{A}_d$ is contained in the behaviour of some variant of $\mathcal{A}_r$.

Definition 4 (The conformance mapping Φ). *Let $\mathcal{A}_r$ and $\mathcal{A}_d$ be a pair of FSMr and FSMd respectively with global predicates ρ^r and ρ^d. Let Π_d, Π_r be the set of all design, requirement configurations. Then $\mathcal{A}_d$ conforms to $\mathcal{A}_r$ if there exists a mapping $\Phi : \Pi_d(\rho^d) \to 2^{\Pi_r(\rho^r)}$ as follows: For any $\pi_d \in \Pi_d(\rho^d)$, $\Phi(\pi_d) = \{\pi_r \in \Pi_r(\rho^r) \mid L(\mathcal{A}_d \downarrow \pi_d) \subseteq L(\mathcal{A}_r \downarrow \pi_r)\}$. Φ is called the conformance mapping, and the conformance via Φ is denoted $\mathcal{A}_d \leq_\Phi \mathcal{A}_r$.*

In the feature *Door lock*, $\Phi(\langle Moff, Speed\rangle)$ contains $\langle Enable, Manual, Speed\rangle$ and $\langle Enable, Auto, Speed\rangle$.

2.3 Checking the Conformance

Let f be a feature with FSMr Req_f and FSMd Des_f. Then the conformance checking problem is to compute a mapping Φ such that $Des_f \leq_\Phi Req_f$.

The conformance mapping is computed by comparing every variant of Des_f with every variant of Req_f. Algorithm 1, given below, presents a possible implementation using the standard automata containment algorithm [33], as implemented in the SPIN model checker [6]. Algorithm 1 runs the full verification algorithm of SPIN for every pair (π_d, π_r) of design and requirement configurations. SPIN(i.e. *pan(.exe)*) returns the list of pairs for which the conformance condition is violated. Every other pair is added to the conformance mapping Φ.

It must be noted that even though we are exhaustively checking whether every variant of the design conforms to some variant of the requirement, we are doing it only at the feature level, and not at the product level. Typically, the number of variants per feature is much smaller than the number of variants in the products. Our experimental results (see Section 4) shows that our approach scales well.

Algorithm 1. implements the conformance checking using SPIN.

```
Input : $Des_f$, $Req_f$.
Output : The mapping $\Phi$ when $Des_f \leq_{\Phi} Req_f$
1. Generate a Promela file which contains $Req_f$, $Des_f$, the environment, the conformance condition expressed as a never claim, and the initialization sequence.
2. Launch the full verification algorithm of SPIN
3. Build the mapping $\Phi$ from the output of SPIN.
4. Conclude whether the design conforms to the requirement
if $\forall \pi_d \in \Pi(\rho_d)$, $\Phi(\pi_d) \neq \emptyset$ then
    return true along with ($\Phi$)
else
    return false along with ($\pi_d$) {where $\pi_d$ has no correspondence through $\Phi$}
end if
```

3 Design Verification of SPL

In the previous section, we looked at individual features in an SPL and provided a method for comparing the design and requirements of a feature, both containing variabilities. In this section, we extend this method to verify a whole SPL design against its requirements. An SPL is essentially a composition of multiple features satisfying certain constraints. We define a parallel composition operator over FSMv to model an SPL. The features in an SPL can interact and we follow one of the standard methods of allowing the composed FSMv models to share some common events, which correspond to two-party handshake communication events. A distinguishing aspect of the proposed parallel operator is that it takes into account the constraints across the composed machines.

Definition 5 (Parallel composition of FSMv)
Let $\mathcal{A}_x = \langle Q_x, q_0^x, \Sigma_x, Var_x, E_x, \rho_x \rangle$, $x \in \{1, 2\}$ be two FSMv's with $Var_1 \cap Var_2 = \emptyset$. Let $H = \Sigma_1 \cap \Sigma_2$ be the set of handshaking events. Let ρ_{12} be a predicate over $Var_1 \cup Var_2$, such that $\rho_{12} \wedge \rho_1 \wedge \rho_2$ is consistent. ρ_{12} is the composition predicate capturing the possible constraints between the variabilities of the two composed features. Let $\rho = \rho_{12} \wedge \rho_1 \wedge \rho_2$.

The parallel composition of $\mathcal{A}_1$ and $\mathcal{A}_2$ denoted by $\mathcal{A} = \mathcal{A}_1 \parallel \mathcal{A}_2$ is a tuple $\langle Q_1 \times Q_2, (q_0^1, q_0^2), \Sigma_1 \cup \Sigma_2, Var_1 \cup Var_2, E, \rho \rangle$ with transitions defined as follows: Consider a state $(s_1, s_2) \in Q_1 \times Q_2$, and transitions $(s_1, g_1, a_1, s_1') \in E_1$ and $(s_2, g_2, a_2, s_2') \in E_2$.
(1) If $a_1 = a_2 = a \in H$, define $((s_1, s_2), g_1 \wedge g_2, a, (s_1', s_2')) \in E$, if $g_1 \wedge g_2$ is consistent. This transition is enabled under a valid configuration $\pi \in \Pi(\rho)$, such that $\pi \models g_1 \wedge g_2$.
(2) If $a_1 \in \Sigma_1 \backslash H$, define $((s_1, s_2), g_1, a_1, (s_1', s_2)) \in E$. This transition is enabled under valid configurations π such that $\pi \models g_1$.
(3) If $a_2 \in \Sigma_2 \backslash H$, define $((s_1, s_2), g_2, a_2, (s_1, s_2')) \in E$. This transition is enabled under valid configurations π such that $\pi \models g_2$.

For illustration, consider the feature *Door unlock* which automates the unlocking of the doors in a vehicle. Figure 4-a gives the FSMr of the feature extracted from the requirements. From the initial state, the feature becomes active when the event *Lock* happens. As soon as either the key is removed from ignition or the gear is shifted to park position, the doors get unlocked and the feature *Door unlock* becomes inactive. Figure 4-b presents the FSMd of the feature *Door unlock*. It is quite similar to the requirement except that the active state is split in two: the feature reacts to the *ignition Off* event in one state, and to the *Shift Into Park* event in another state. Let us consider the composition of the two FSMr's of the features *Door lock* and *Door unlock*. The handshake events between the two features are *Lock* and *Unlock*. In the composition, we introduce the following composition predicate: $(DU\_Enable = Enable \Leftrightarrow DL\_Enable = Enable) \wedge Transmission_{dl} = Transmission_{du}$, which brings out the natural constraints that *Door lock* feature is enabled if and only if *Door unlock* is also enabled and the transmission status has to be the same. The valid configurations after composition are restricted by the composition predicate. We provide a few definitions to define composite valid configurations.

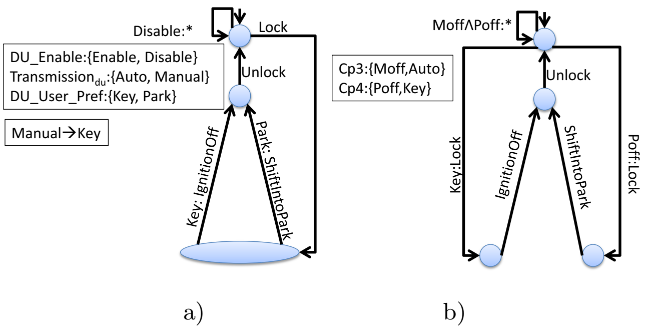


Fig. 4. a) Req_{du}: *Door unlock* FSMr and b) Des_{du}: *Door unlock* FSMd

Definition 6 (Composing Configurations). *Let* $\mathcal{A}_i = (Q_i, q_0^i, \Sigma_i, Var_i, E_i, \rho_i)$ *be two FSMv's,* $i = 1, 2$*, and let* $\mathcal{A} = \mathcal{A}_1 \parallel \mathcal{A}_2$ *be as given by Definition 5. Let* $\rho = \rho_{12} \wedge \rho_1 \wedge \rho_2$ *be the global predicate of* $\mathcal{A}$*. Consider two valid configurations* $\pi_1 \in \Pi(\rho_1)$ *and* $\pi_2 \in \Pi(\rho_2)$ *of* $\mathcal{A}_1$ *and* $\mathcal{A}_2$*. The compostion of* π_1, π_2 *denoted* $\pi_1 + \pi_2$ *is a configuration over* $Var_1 \cup Var_2$ *such that (i)* $\pi_1 + \pi_2$ *agrees with* π_1 *over* Var_1*, agrees with* π_2 *over* Var_2*, and (ii)* $\pi_1 + \pi_2 \models \rho$*.*

Lemma 7. *Let* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *be two FSMv's. For each valid configuration* π *of* $\mathcal{A}_1 \parallel \mathcal{A}_2$*, there are valid configurations* π_1 *of* $\mathcal{A}_1$ *and* π_2 *of* $\mathcal{A}_2$ *such that* $\pi = \pi_1 + \pi_2$*.*

Due to lack of space, proofs have been omitted. Proofs of all the results can be found in [34].

In the example of feature *Door Lock*, the configuration $\langle Enable, Auto, Speed\rangle$ from Req_{dl} can be composed with $\langle Enable,\ Auto, Key\rangle$ from Req_{du} because the transmission is *Auto* in both (which is specified in the composition predicate ($DU\_Enable = Enable \Leftrightarrow DL\_Enable = Enable) \wedge Transmission_{dl} = Transmission_{du}$). $\langle Enable, Auto, Speed, Enable, Auto, Key\rangle$ is a configuration of the parallel composition of Req_{dl} with Req_{du}. The parallel composition of FSMv's is such that each variant of the composition of two FSMv's is equal to the composition of variants of the individual FSMv's.

Lemma 8 (Variants of a composed FSMv). *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two FSMv's. Let π be a valid configuration of $\mathcal{A}_1 \parallel \mathcal{A}_2$. Then $L([\mathcal{A}_1 \parallel \mathcal{A}_2] \downarrow \pi) = L(\mathcal{A}_1 \downarrow \pi) \parallel L(\mathcal{A}_2 \downarrow \pi)$.* [1]

Refinement and Parallel Composition. The definition of parallel composition naturally lends itself to a notion of addition of conformance mappings between design and requirement pairs. Consider FSMr's R_1, R_2 corresponding to two features f_1, f_2. Let D_1, D_2 be the corresponding FSMd's. Let ρ^r_1, ρ^r_2 be the global predicates of R_1, R_2, and let ρ^d_1, ρ^d_2 be the global predicates of D_1, D_2 respectively. Assume that $D_1 \leq_{\Phi_1} R_1$ and $D_2 \leq_{\Phi_2} R_2$. Let $\rho^r = \rho^r_{12} \wedge \rho^r_1 \wedge \rho^r_2$ be the global predicate of $R_1 \parallel R_2$; likewise, let $\rho^d = \rho^d_{12} \wedge \rho^d_1 \wedge \rho^d_2$ be the global predicate of $D_1 \parallel D_2$. We now want to ask if $D_1 \parallel D_2$ conforms to $R_1 \parallel R_2$. This amounts to computing a conformance mapping between $D_1 \parallel D_2$ and $R_1 \parallel R_2$ given Φ_1, Φ_2. Consider any valid configuration π^d of $D_1 \parallel D_2$. By Lemma 7, we can write π^d as $\pi^d_1 + \pi^d_2$, where π^d_1, π^d_2 are valid configurations of D_1, D_2 respectively. Since $D_1 \leq_{\Phi_1} R_1$ and $D_2 \leq_{\Phi_2} R_2$, there exists valid configurations $\pi^r_1 \in \Phi_1(\pi^d_1)$ and $\pi^r_2 \in \Phi_2(\pi^d_2)$ in R_1, R_2 respectively. Given this, the addition of Φ_1, Φ_2 is defined as follows:

Definition 9 (Addition of conformance mappings). *The addition of conformance mappings Φ_1, Φ_2 is defined to be a mapping $\Phi = \Phi_1 + \Phi_2$ as follows. For every valid configuration $\pi^d = \pi^d_1 + \pi^d_2$ of $D_1 \parallel D_2$,*

$$\Phi(\pi^d) = \{\pi^r \mid \pi^r \textit{ is a valid configuration of } R_1 \parallel R_2, \pi^r = \pi^r_1 + \pi^r_2 \\ \textit{for valid configurations } \pi^r_1 \in \Phi_1(\pi^d_1), \pi^r_2 \in \Phi_2(\pi^d_2)\}$$

Note that by Definition 9, Φ could be empty: Consider a valid configuration $\pi^d = \pi^d_1 + \pi^d_2$ of $D_1 \parallel D_2$. If there is no valid configuration π^r of $R_1 \parallel R_2$ which is a composition of valid configurations $\pi^r_1 \in \Phi_1(\pi^d_1), \pi^r_2 \in \Phi_2(\pi^d_2)$, then Φ is empty (or there is no conformance mapping Φ between $D_1 \parallel D_2$ and $R_1 \parallel R_2$). If Φ exists, then we can say the following:

Lemma 10 (Conformance of composition). *Let R_1 and R_2 be two FSMrs corresponding to features f_1, f_2, and let D_1 and D_2 be the corresponding FSMds. Let $D_1 \leq_{\Phi_1} R_1$ and $D_2 \leq_{\Phi_2} R_2$. Let $\Phi = \Phi_1 + \Phi_2$ and π^d be a valid configuration of $D_1 \parallel D_2$. Then, $\forall \pi^r \in \Phi(\pi^d)$, $L([(D_1 \parallel D_2) \downarrow \pi^d]) \subseteq L([(R_1 \parallel R_2) \downarrow \pi^r])$.*

[1] The right hand side $\parallel$ refers to the standard communicating finite state machine composition.

Considering the example, in the FSMr $Req_{dl} \parallel Req_{du}$ with $\rho_r : DL\_Enable = DU\_Enable \wedge Transmission_{dl} = Transmission_{du}$, any configuration having $DL\_Enable{=}Enable$ and $DU\_Enable = Disable$ is invalid. However, $\Phi(\langle Auto, Speed\rangle)$ contains only configurations where $DL\_Enable = Enable$, $\Phi'(\langle Moff, Poff\rangle)$ contains only configurations where $DU\_Enable = Disable$ and $\langle Auto, Speed\rangle + \langle Moff, Poff\rangle$ is a valid configuration of $Des_{dl} \parallel Des_{du}$. So the design does not conform to the requirement. However, if we consider the composition predicate $\rho_d : Cp1 = Moff \wedge Cp2 = Poff \Leftrightarrow Cp3 = Moff \wedge Cp4 = Poff$, then $\langle Auto, Speed\rangle$ and $\langle Moff, Poff\rangle$ are not compatible anymore and as a result the design conforms to the requirement.

3.1 Conformance Checking

Consider the case when we have n features $f_1, \ldots, f_n$, with FSMds $D_1, \ldots, D_n$ and FSMrs $R_1, \ldots R_n$, such that $D_i \leq_{\Phi_i} R_i$ for $1 \leq i \leq n$. When can we say that $D_1 \parallel \cdots \parallel D_n$ conforms to $R_1 \parallel \cdots \parallel R_n$? For all valid design configurations $\pi_{D_1 \parallel \cdots \parallel D_n}$ of $D_1 \parallel \cdots \parallel D_n$, we check the existence of a configuration $\pi_{R_1 \parallel \cdots \parallel R_n}$ of $R_1 \parallel \cdots \parallel R_n$ such that $\pi_{R_1 \parallel \cdots \parallel R_n}$ is a composition of valid requirement configurations computed via $\Phi_1, \ldots, \Phi_n$. We then say $D_1 \parallel \cdots \parallel D_n$ conforms to $R_1 \parallel \cdots \parallel R_n$ via a conformance mapping Φ. It can be observed that Φ is nothing but $\Phi_1 + \cdots + \Phi_n$. We now formulate the existence of a conformance mapping Φ using a QBF.

QBF Formulation. Given FSMd's $D_1, \ldots, D_n$ and FSMr's $R_1, \ldots, R_n$,
(1) Let $Var(D_i) = \{x_{i1}, \ldots, x_{id_i}\}$ be the set of variables of design D_i, and $Var(R_i) = \{y_{i1}, \ldots, y_{ir_i}\}$, the set of variables of requirement R_i. Let $\pi_i^d : (x_{i1} = a_1, \ldots, x_{id_i} = a_{d_i})$ be a configuration of D_i. We denote this by $\pi_i^d(x_{i1}, \ldots, x_{id_i})$, which is the conjunction $\bigwedge_{l=1}^{d_i}(x_{il} = a_l)$;
(2) Given n FSMd's and n FSMr's check if D_i conforms to R_i for all $1 \leq i \leq n$ using Algorithm 1. This gives the map Φ_i. Assume D_i has m distinct configurations $\pi_{i1}^d, \ldots, \pi_{im}^d$. For $1 \leq j \leq m$, let $\Phi_i(\pi_{ij}^d) = \{\pi_{ij_1}^r, \ldots, \pi_{ij_k}^r\}$, where each of $\pi_{ij_1}^r, \ldots, \pi_{ij_k}^r$ are configurations of R_i, that have been mapped by Φ_i to some configuration π_{ij}^d of D_i. $\Phi_i(\pi_{ij}^d)$ can be written as the formula $\pi_{ij_1}^r \vee \cdots \vee \pi_{ij_k}^r$.
(3) The conformance mapping Φ_i between D_i and R_i then has the form $\bigwedge_{j=1}^{m} \Phi_i(\pi_{ij}^d)$. (4) Let $\varphi_{i,j}^d = \rho^d \wedge \rho_i^d \wedge \rho_j^d$ and $\varphi_{i,j}^r = \rho^r \wedge \rho_i^r \wedge \rho_j^r$ represent respectively the propositional formulae which ensure the consistency of the global predicates of D_i, D_j and R_i, R_j along with the compositional predicates ρ^d and ρ^r. Given a set $S \subseteq \{1, 2, \ldots, n\}$, φ_S^d and φ_S^r can be appropriately written.
The QBF for conformance checking is given by

$$\Psi = \forall x_{11} \ldots x_{1d_1} x_{21} \ldots x_{2d_2} \ldots x_{n1} \ldots x_{nd_n} [\varphi_{1,2,\ldots,n}^d \Rightarrow \\ \exists y_{11} \ldots y_{1r_1} y_{21} \ldots y_{2r_2} \ldots y_{n1} \ldots y_{nr_n} (\Phi_1 \wedge \cdots \wedge \Phi_n \wedge \varphi_{1,2,\ldots,n}^r)]$$

The theorem below asserts that the QBF Ψ is true iff a conformance mapping Φ exists such that $D_1 \parallel \cdots \parallel D_n \leq_{\Phi} R_1 \parallel \cdots \parallel R_n$.

Theorem 1. *Given an SPL, let $\{f_1, \ldots, f_n\}$ be the set of features in a chosen product. Let D_i, R_i be the FSMd and FSMr for feature f_i. Then $D_1 \parallel \cdots \parallel D_n$ conforms to $R_1 \parallel \cdots \parallel R_n$ iff Ψ, as defined above, holds.*

4 Implementation and Case Studies

Our prototype tool SPLEnD, takes as input pairs of XML files corresponding to FSMd, FSMr and outputs a PROMELA file. The latter is fed to SPIN, which returns the conformance mappings, or declares non-conformance. On the given conformance mapping, the tool computes a QBF Ψ which is fed to CirQit. The experiments were run on a 2.24 GHz i3 processor machine with 3GB RAM.

Features	*PL & LDCL*	*PCU*	*DL*	*DU*	*AL*	*TSL*
Design Variants	8	3	4	7	3	8
SPIN Time (Sec)	0.436	0.031	0.046	0.109	0.015	0.218

Fig. 5. Execution time of FSMv-Verifier on Algorithm 1 for ECPL

Features(Design Variants)	Time(ms)	Features(Design Variants)	Time(ms)
UserInterface(6)	2	CheckingBalance(3)	3
WithdrawMoney(8)	27	DepositMoney(2)	2
PrintingStatement(3)	2	Login(1)	1
ATMLogin(1)	1	ChangeAccountPassword(2)	3
PayBills(2)	3	PrintingBalanceAfterWithdraw(2)	3
CheckingMoneyExchangeRate(2)	3	MoneyExchange(2)	4
InternationalTransfer(2)	6	LocalTransferToOtherBank(1)	4
LanguageSelection(2)	1	MobileTopUp(2)	2
ChangeMaxLimitForWithdrawal(1)	3	LocalTransferToSameBank(3)	3
AddBeneficiary(1)	2	RemoveBeneficiary(1)	2
CreateDemandDraft(2)	3	ChequeClearance(1)	3
FastWithdrawal(1)	2	CreditCardPayment(2)	2
UpdateContactDetails(2)	4	RegisterMobileNoForBanking(2)	2
OpenAccount(8)	30	CloseAccount(2)	5
ActivateAccount(2)	4	ReactivateAccount(2)	4

Fig. 6. Execution time of SPLEnD on Algorithm 1 for BSPL

We considered two real case studies for our experimentation: Entry Control Product Line, ECPL having 7 features and Banking Software Product Line, BSPL, composed of 30 features. In an earlier study [34], we considered BSPL with 25 features; in this paper, we consider an enhanced version of BSPL by adding 5 more features. The FSMr, FSMd models of each feature contain less than 15 states. The analysis results for the two case studies are summarized in Figures 5 and 6 which gives the times taken by Algorithm 1. The number of variants per feature is at most 8 in both cases. In the case of ECPL, a non-conformance was found in the feature *Door Lock* [2]. For BSPL, the second step using the QBF approach and CirQit took just 0.022 seconds. Encouraged by this result, we are currently looking at some large industrial case studies.

References

1. Benavides, D., Segura, S., Cortés, A.R.: Automated analysis of feature models 20 years later: A literature review. Inf. Syst. 35(6), 615–636 (2010)
2. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A.: Symbolic model checking of software product lines. In: ICSE, pp. 321–330 (2011)
3. Cordy, M., Classen, A., Perrouin, G., Schobbens, P.Y., Heymans, P., Legay, A.: Simulation-based abstractions for software product-line model checking. In: ICSE, pp. 672–682 (2012)
4. Czarnecki, K., Eisenecker, U.W.: Generative programming - methods, tools and applications. Addison-Wesley (2000)
5. Metzger, A., Pohl, K.: Variability management in software product line engineering. In: ICSE Companion, pp. 186–187 (2007)
6. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional (2003)
7. Goultiaeva, A., Bacchus, F.: Exploiting qbf duality on a circuit representation. In: AAAI (2010)
8. Apel, S., Speidel, H., Wendler, P., Rhein, A., Beyer, D.: Detection of feature interactions using feature-aware verification. In: ASE, pp. 372–375 (2011)
9. Apel, S., Hutchins, D.: A calculus for uniform feature composition. ACM Trans. Program. Lang. Syst. 32(5) (2010)
10. Harry, C., Li, S.K., Fisler, K.: Verifying cross-cutting features as open systems. In: Daemen, J., Rijmen, V. (eds.) FSE 2002. LNCS, vol. 2365, pp. 89–98. Springer, Heidelberg (2002)
11. Fischbein, D., Uchitel, S., Braberman, V.: A foundation for behavioural conformance in software product line architectures. In: ROSATEA, pp. 39–48 (2006)
12. Asirelli, P., Maurice, H., terBeek, S.G., Fantechi, A.: Formal description of variability in product line families. In: SPLC, pp. 130–139 (2011)
13. Schaefer, I., Gurov, D., Soleimanifard, S.: Compositional algorithmic verification of software product lines. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 184–203. Springer, Heidelberg (2011)
14. Gondal, A., Poppleton, M., Butler, M.: Composing event-b specifications - case study experience. In: Apel, S., Jackson, E. (eds.) SC 2011. LNCS, vol. 6708, pp. 100–115. Springer, Heidelberg (2011)
15. Mannion, M.: Using first-order logic for product line model validation. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379, pp. 176–187. Springer, Heidelberg (2002)
16. Batory, D.: Feature models, grammars, and propositional formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)

[2] In Des_{dl}, the transition from the middle elliptical state to the round state labeled with $Poff : ShiftOutOfPark$ is incorrect; $\Phi(\langle Auto, Poff\rangle) = \emptyset$. Removing this transition fixes the bug.

17. Larsen, K.G., Nyman, U., Wąsowski, A.: Modal I/O automata for interface and product line theories. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 64–79. Springer, Heidelberg (2007)
18. Raclet, J.B., Badouel, E., Benveniste, A., Caillaud, B., Legay, A., Passerone, R.: Modal interfaces: unifying interface automata and modal specifications. In: EMSOFT, pp. 87–96 (2009)
19. Fantechi, A., Gnesi, S.: Formal modeling for product families engineering. In: SPLC 2008, pp. 193–202. IEEE Computer Society (2008)
20. Gruler, A., Leucker, M., Scheidemann, K.: Calculating and modeling common parts of software product lines. In: SPLC, pp. 203–212 (2008)
21. Gomaa, H., Olimpiew, E.M.: Managing variability in reusable requirement models for software product lines. In: Mei, H. (ed.) ICSR 2008. LNCS, vol. 5030, pp. 182–185. Springer, Heidelberg (2008)
22. Jörges, S., Lamprecht, A.L., Margaria, T., Schaefer, I., Steffen, B.: A constraint-based variability modeling framework. In: STTT, vol. 14(5), pp. 511–530 (2012)
23. Berg, K., Bishop, J., Muthig, D.: Tracing software product line variability: from problem to solution space. In: Proceedings of the 2005 Annual Research Conference on IT Research in Developing Countries, SAICSIT 2005, pp. 182–191 (2005)
24. Metzger, A., Heymans, P., Pohl, K., Schobbens, P.Y., Saval, G.: Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In: RE, pp. 243–253 (2007)
25. Riebisch, M., Brcina, R.: Optimizing design for variability using traceability links. In: ECBS, pp. 235–244 (2008)
26. ter Beek, M.H., Mazzanti, F., Sulova, A.: VMC: A Tool for product variability analysis. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 450–454. Springer, Heidelberg (2012)
27. ter Beek, M.H., Gnesi, S., Mazzanti, F.: Demonstration of a model checker for the analysis of product variability. In: SPLC, pp. 242–245 (2012)
28. Krishnamurthi, S., Fisler, K.: Foundations of incremental aspect model-checking. ACM Trans. Softw. Eng. Methodol. 16(2) (2007)
29. Liu, J., Basu, S., Lutz, R.R.: Compositional model checking of software product lines using variation point obligations. Autom. Softw. Eng. 18(1), 39–76 (2011)
30. Cordy, M., Schobbens, P.Y., Heymans, P., Legay, A.: Behavioural modelling and verification of real-time software product lines. In: SPLC, vol. 1, pp. 66–75 (2012)
31. Lauenroth, K., Metzger, A., Pohl, K.: Quality assurance in the presence of variability. Technical report, SSE, Institut fur Informatik und Wirtschaftsinformatik, univertitat Duisburg Essen (2011)
32. Gruler, A., Leucker, M., Scheidemann, K.: Modeling and model checking software product lines. In: Proceedings of the 10th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (2008)
33. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proceedings of LICS 1986, pp. 322–331 (1986)
34. Millo, J.V., Ramesh, S., Krishna, S.N., Narwane, G.K.: Compositional verification of evolving software product lines. CoRR abs/1212.4258 (2012)

Deductive Verification of State-Space Algorithms

Frédéric Gava, Jean Fortin, and Michael Guedj

Laboratory of Algorithms, Complexity and Logic (LACL), University of Paris-East
{frederic.gava,jean.fortin,michael.guedj}@univ-paris-est.fr

Abstract. As any software, model-checkers are subject to bugs. They can thus report false negatives or validate a model that they should not. Different methods, such as theorem provers or Proof-Carrying Code, have been used to gain more confidence in the results of model-checkers. In this paper, we focus on using a verification condition generator that takes annotated algorithms and ensures their termination and correctness. We study four algorithms (three sequential and one distributed) of state-space construction as a first step towards mechanically-assisted deductive verification of model-checkers.

Keywords: BSP, Model-checking, Deductive verification, State-space.

1 Introduction

Motivation. Model-checkers (MCs for short) are often used to verify safety-critical systems. The correctness of their answers is thus vital: many MCs produce the answer "yes" or generate a counterexample computation (if a property of the model fails), which forces, in the two cases, to assume that the algorithm and its implementation are both correct.

But MCs, like any software are subject to bugs and there exist surprisingly few attempts to prove them correct. Three main reasons can explain this fact [13]: (1) MCs involve complicated logics, algorithms and sophisticated state reduction techniques; (2) because efficiency is essential, MCs are often highly optimised, which implies that they may not be designed to be proved correct; (3) MCs are often updated. But there is a more and more pressing need from the industrial community, as well as from national authorities, to get not just a boolean answer, but also a formal proof — which could be checked by an established tool such as the theorem prover Coq. This is required in Common Criteria certification of computer products at the highest assurance level EAL 7 — `http://www.commoncriteriaportal.org/`. And hand proofs are not sufficient for EAL 7, mechanical proofs are needed. The author of [18] resumes the problem: *Quis custodiet ipsos custodes ?* (Who will watch the watchmen? that is, who will verify the verifier?). We want to be able to trust the results of model-checkers with a high degree of confidence.

Different Solutions for Verifying Model-Checkers. For verifying model-checkers, different solutions have been proposed. The first one is to prove MCs inside theorem provers and use the extraction facilities to get pure functional machine-checked programs such as in the works of [20] and [6]. The second and more common approach, in the spirit of Proof-Carrying Code [14] (PCC for short), is to generate a "certificate" during the execution of the MC that can be checked later or on-the-fly by a dedicated tool or a theorem prover. This is the so-called "certifying model-checking" [13]. In this way, users can re-execute the certificate/trace and have some safety guarantees because even if the MC is buggy, its results can be checked by a trustworthy tool.

But, any explicit MC may enumerate a very large state-spaces (the famous state-space explosion problem), and mimicking this enumeration with proof rules inside any theorem prover (or with PCCs) would be foolish even if specific techniques and optimisations of the abstract machine of theorem provers [1] are used. Note that this problem does not arise when finding a refutation of the logical formula (the trace is generally short) but when the answer is "yes" since the entire explicit state-space (or at least a symbolic representation) needs to verify the checked properties. In this way, certificate generation could also hamstring both the functionality and the efficiency of the automation that can be built from theorem provers (functional programs can be too memory consuming) and PCC tools (too big certificates) [18]. Only efficient, imperative and distributed programs can override the state-space explosion problem.

Another solution, proposed in [22] for a MC call PAT, is to use coding assumptions directly in the source code. They indeed use Spec# and a check of the object invariants (the contracts) is generated. Nevertheless, they cannot completely verify the correctness of PAT and they thus focus on some safety properties (as no overflows, no deadlocks) of the underlying data structures of PAT (which can run on a multi-core architecture) and check if some options may conflict with each other.

The Proposed Solution and Outline. Our contribution follows the approach of [22] but by using the "verification condition generator" (VCG for short) *WHY* [7] and by extending the verification to the correctness of the final result: has the full state-space been well computed without adding unknown states?

Since the language of *WHY* is not immediately executable but a higher-level algorithmic language, we only focus on algorithms. We can thus focus on which formal properties need to be preserved and not be obstructed by problems specific to a particular programming language. Even if most of the bugs in MCs will not be due to wrong algorithms but rather due to subtle errors in the implementation of some complex data structures and bad interactions between these structures and compression aspects, we must first check the algorithms to get an idea of the amount of work necessary to verify a true model-checker.

Our goal is then a mechanically-assisted proof that these annotated algorithms terminate and indeed compute the expected finite state-space. This is an interesting first step before verifying MCs themselves: it allows to test if this approach is doable or not. This is also challenging due to the nature of model-checking (critical system) and to the algorithmic complexity. The main contribution of this paper is to demonstrate the ability of a VCG such as *WHY* to tackle the wide range of verification issues involved in the proof of correctness of imperative codes of MCs.

The remainder of this paper is structured as follows. The VCG *WHY* is presented in Section 2.1, then the full state-space if formally defined in Section 2.2; we consider also verifying different algorithms formally: three sequential ones (which correspond to those mainly used in explicit MCs; described in Section 2.3; verified in Section 2.4) and one distributed — mainly used in explicit distributed MCs; described in Section 3.3; verified in Section 3.4. The first three are relatively simple to prove correct: it is thus a good basis for correctness of MCs. For the last, we use our own extension of *WHY* called *BSP-WHY* [9], which is presented in Section 3.2. Section 4 discusses some related work and finally, Section 5 concludes the paper and gives a brief outlook to future work.

2 Verification of Sequential State-Space Algorithms

We now introduce the VCG *WHY*, describe how we model the state space, and present the verification of 3 well-known algorithms. The annotated source codes are available at `http://lacl.fr/gava/cert-mc.tar.gz`.

2.1 Deductive Verification of Algorithms Using *WHY*

WHY [7] is a framework for the verification of algorithms. Basically, it is composed of two parts: a logical language with an infrastructure to translate it to existing theorem provers; and an intermediate verification programming language called *WhyML* with a VCG for deductive verification. The logic of *WHY* is a polymorphic first-order logic with logical declarations: definitions and axioms. The examples of the standard library propose finite sets of data and several operations with their axiomatisation (which can be proved using Coq): a constant empty set; functions add, remove, union, inter, diff, cardinal; a predicate for emptiness, equality, subset, extensionality, *etc.* In the logical formula, x@ is the notation for the value of x in the pre-state, *i.*e. at the precondition point and x@label for the value of x at a certain point (marked by a label) of the algorithm.

WhyML is a first-order language with an ML flavored syntax and it provides the usual constructs of imperative programming. All symbols from the logic can be used in the algorithms. Mutable data types can be introduced, by means of polymorphic references: a reference r to a value of type σ has type **ref** σ, is created with the function **ref**, is accessed with !r, and assigned with r $\leftarrow$e. Algorithms are annotated using pre- and post-conditions, loop invariants, and variants to ensure termination. Verification conditions are computed using a weakest precondition (wp) calculus and then passed to the back-end of *Why* to be sent to provers. Notice that in *WHY*, sets are immutable (manipulated only with purely functional routines) and thus only a reference on a set can be modified and assigned to another set.

```
1 let normal () =
2   let known = ref ∅ in
3   let todo = ref {s0} in
4    while todo ≠ ∅ do
5     let s = pick todo in
6       known←!known ⊕ s;
7       todo←!todo ∪ (succ(s) \ !known)
8    done;
9    !known
```

```
1  let main_dfs () =
2   let known = ref ∅ in
3   let rec dfs (s:state) : unit =
4     known←!known ⊕ s;
5     let current = ref (succ(s) \ !known) in
6      while current ≠ ∅ do
7        let new_s = pick current in
8          if (new_s ∉ known) then dfs(new_s)
9      done;
10   in dfs(s0); !known
```

Fig. 1. Sequential WhyML algorithms

2.2 Definition of the Finite State-Space

Let us recall that the finite state-space construction problem is computing the explicit graph representation (also known as *Kripke structure*) of a given model from the implicit one. This graph is constructed by exploring all the states reachable through a successor function succ (which returns a set of states) from an initial state s_0. Generally, during this operation, all the explored states must be kept in memory in order to avoid multiple explorations of a same state.

In this paper, all algorithms only compute the state-space, noted StSpace. This is done without loss of generality and it is a trivial extension to compute the full Kripke structure — usually preferred for checking temporal logic formulas. To represent StSpace in the logic of *WHY*, we used the following axiom contain_state_space (for consistency, it has been proved in Coq using an inductive definition of the state-space, also available in the source code):

```
logic s0: state      logic succ: state → state set      logic StSpace: state set
axiom contain_state_space: ∀ss:state set. StSpace ⊆ ss ↔
                                         (s0 ∈ ss and (∀ s:state. s ∈ ss → s ∈ StSpace → succ(s) ⊆ ss))
```

i.e. defines which sets can contain the state-space. Now ss is the state-space (ss=StSpace) if and only if, the two following properties holds: (A) ss $\subseteq$ StSpace and (B) StSpace $\subseteq$ ss; that is equality of sets using extensionality. Note that using this first-order definition makes the automatic (mainly SMT) solvers prove more proof obligations than using an inductive definition for the state-space.

2.3 Sequential Algorithms for State-Space Construction

Fig. 1 gives two common algorithms in WhyML using an appropriate syntax for set operations — a "Breadth-first" algorithm is also fully available in the source code but not presented here due to lack of space. All computations in these programs are set operations where a set call known contains all the states that have been processed and would finally contain StSpace.

The first one, called "Normal", corresponds to the usual sequential construction of a state-space —random walk. It involves a set of states todo that is used to hold all the states whose successors have not been constructed yet; each state s from todo is processed in turn (lines $4-5$) and added to known (line 6) while its successors are added to todo unless they are known already — line 7.

The second one is the standard recursive algorithm "Dfs". At each call of dfs(s), the state s is added (side-effect) to known (line 3) and dfs is then recursively called (lines 5 – 8) for all the successors of s unless they are already known — which is an optimization since these states would anyway be filtered out later on. Note the use of a conditional (line 8) within this loop: this is due to the fact that during the exploration of the successors of s, known can be increased and thus this prevents the re-exploration of these states

Note that the "Normal" algorithm can be made strictly depth-first by choosing the most-recently discovered state (*i.e.* todo as a stack), and breadth-first by choosing the least-recently discovered state. This has not been studied here.

2.4 Verification of These Algorithms

For correctness, the previously presented codes need three properties: (1) they do not fail (no rule of reduction); (2) they indeed compute the state-space; (3) and they terminate. The first property is immediate since the only operation that could fail is pick (where the precondition is "not take any element from an empty set") and this is assured by the guard of the **while** loop. Let us now focus on the specification of the above algorithms.

Annotations. Fig. 2 gives the full annotated code of the "Dfs" algorithm and "Normal" needs only adding the following invariants in the loop (and final post-condition {result=StSpace}):

```
invariant (1) (known ∪ todo) ⊆ StSpace
      and (2) (known ∩ todo)=∅
      and (3) s0 ∈(known ∪ todo)
      and (4) (∀ e:state. e ∈known → succ(e) ⊆ (known ∪ todo))
   variant |StSpace \ known|
```

These four invariants are: (1) known and todo are subsets of StSpace; at the end, (3) and (4) known is a subset of StSpace and has the "same" inductive property; and when todo will be empty, then known contains StSpace — property (B).

"Dfs" is more subtle. We need to introduce ghost codes[1], notably a set nofinish (line 3) which has the following rule: each state s in nofinish has been processed by the dfs function but not completely that is, s is in known and not all its direct successors have been processed by dfs — in the loop. It is used in the pre-condition (lines 8-9) and post-condition (lines 31-34) of dfs since not all the direct successors have been processed since it is a depth-first algorithm.

Also nofinish is a subset of known since all the time, each state s will be finally completely processed. That also forces us to add this fact in pre- and post-conditions. The post-conditions (1) and (2) are used for (A) and (B). Note the use of nofinish since some states can not be fully processed but nofinish is empty at the end of the computation, ensuring (B). The two post-conditions (5) and (7) say that nofinish is the same before and after dfs (thus empty when s0 is fully processed) but known was able to increase.

Now the invariants (lines 18−22) of the loop are the following: (1) and (2) as in "Normal", the set known is a subset of StSpace (current is the set succ(s)−known

[1] Additional codes not participating in the computation but accessing the program data and allowing the verification of the original code.

```
 1 let main_dfs () =
 2   let known = ref ∅ in
 3   let nofinish = ref ∅ in (* ghost *)
 4   let rec dfs (s:state) : unit
 5    variant |Stspace \ known|
 6    =
 7     {
 8           (1) s ∈StSpace and (2) known ⊆ StSpace and (3) s ∉ known and (4) s ∉ nofinish
 9      and (5) (∀ e:state. e ∈known→ ¬(e ∈nofinish)→ succ(e) ⊆ known) and (6) nofinish ⊆ known
10     }
11     known←!known ⊕ s;
12     nofinish←!nofinish ⊕ s;
13     let current = ref (succ(s) \ !known) in
14     let ghost_diff=ref ∅ in
15      L:while current ≠ ∅ do
16       {
17        invariant
18        (1) (known ∪ current) ⊆ StSpace
19    and (2) (∀ e:state. e ∈known→ ¬(e ∈nofinish)→ succ(e) ⊆ known)
20    and (3) succ(s) ⊆ (known ∪ current) and (4) known@L ⊆ known
21    and (5) current@L= (ghost_diff ∪ current) and (6) (ghost_diff ∩ current)=∅
22    and (7) nofinish=nofinish@L and (8) nofinish ⊆ known
23       variant |current|
24       }
25        let new_s = pick current in
26         ghost_diff←!ghost_diff ⊕ new_s;
27         if (new_s ∉ known) then dfs(new_s)
28      done;
29      nofinish←!nofinish ⊖ s
30      {
31           (1) known ⊆ StSpace
32        and (2) (∀ e:state. e ∈known → ¬(e ∈nofinish) → succ(e) ⊆ known)
33        and (3) s ∈known and (4) s ∉ nofinish and (5) nofinish=nofinish@
34        and (6) known@ ⊆ known and (7) nofinish ⊆ known
35      }
36   in dfs(s0); !known {result=StSpace}
```

Fig. 2. "Dfs" sequential annoted algorithm

used in the foreach statement) and known works as StSpace; (3) all the direct successors of s are in known or are currently processed; (4) known can increase; (5−6) current works well as an iteration over a set using a ghost set which ensures that no elements are lost during the iteration; (7) nofinish is not modified by the loop but before the loop (and the post-condition ensures that it returns as in the beginning of dfs); (8) nofinish remains a subset of known.

Termination. For all the algorithms, termination is ensured by the following variants: $|\mathsf{StSpace} \setminus \mathsf{known}|$ and by $|\mathsf{current}|$ when an iteration on each state of a set is performed. Each algorithm ensures this first variant at every step using the following properties:

- "Normal" only adds a new state s since $(\mathsf{known} \cap \mathsf{todo}) = \emptyset$;
- "Dfs" only recursively adds a new state (line 29) since the pre-condition of the function is $\mathsf{s} \notin \mathsf{known}$ (line 8) and the boolean condition of the conditional is $\mathsf{new\_s} \notin \mathsf{known}$ in the loop for the successors;

Mechanical Proof. All the obligations produced by the VCG of *WHY* are automatically discharged by a combination of automatic provers: CVC3, Z3, Simplify, Alt-Ergo, Yices and Vampire. For each prover, we give a timeout of 10 seconds — otherwise some obligations are not proved. In the following table, for each algorithm, we give the number of generated obligations (column Total) and then how many are discharged by the provers:

algo/Solvers	Total	Alt-Ergo	Simplify	Z3	CVC3	Yices	Vampire
Normal	11	2	10	11	7	3	3
Breadth	31	9	31	28	21	10	10
Dfs	49	22	48	47	40	23	26

One could notice that the SMT solvers Simplify and Z3 give the best results. In practice, we mostly used them. Simplify is the faster and Z3 sometime verified some obligations that had not be discharged by Simplify. We also have worked with the provers as black-boxes and we have thus no explanation for this fact. It also took few days for the first author to annotate all the algorithms. Proof obligations are as usual when working with a VCG such as *WHY*.

3 Verification of a Distributed State-Space Algorithm

Parallelize the construction of the state-space on several machines is a standard method [2,11]. In this section, we give an example of how to verify a parallel algorithm and show that it is more challenging but feasible. We first present our model of parallel computation called BSP then our own extension of *WHY* for BSP algorithms and finally the verification of a BSP state-space algorithm.

3.1 The Bulk-Synchronous Parallel (BSP) Model

In the BSP model, a computer is a set of uniform processor-memory pairs and a communication network allowing the inter-processor delivery of messages [19,4].

A BSP program is executed as a sequence of *super-steps*, each one divided into three successive disjoint phases: each processor only uses its local data to perform sequential computations and to request data transfers to other nodes; the network delivers the requested data; a global synchronisation barrier occurs, making the transferred data available for the next super-step. The BSP model considers communications *en masse* — as MPI's collective operations, Message Passing Interface `http://www.mpi-forum.org/`. This is less flexible than asynchronous messages, but easier to debug and prove since interactions of simultaneous communication actions are typically complex.

3.2 Deductive Verification of BSP Algorithms

Our tool BSP-*WHY* extends the syntax of *WhyML* with BSP primitives (message passing and synchronisation) and definitions of collective operations. BSP-*WhyML* codes are written in a Single Program Multiple Data (SPMD) fashion. We used the *WhyML* language as a back-end of our own BSP-*WhyML* language.

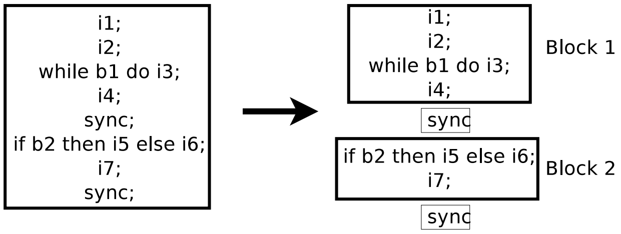


Fig. 3. Example of the BSP-*WHY*'s block decomposition of a BSP code

In this way, BSP-*WhyML* programs are transformed into *WhyML* ones and then the VCG of *WHY* is used to generated the appropriate conditions for the deductive verification of the BSP algorithm.

A special constant **nprocs** (equal to **p** the number of processors) and a special variable **bsp_pid** (with range $0, \ldots, \mathbf{p}-1$) were also added to *WhyML* expressions. A special syntax for BSP annotations is also provided which is simple to use and seems sufficient to express conditions in most practical programs: we add the construct $t<i>$ which denotes the value of a term t at processor id i, and $<x>$ denotes a **p**-value x (represented by *fparray*, purely applicative arrays of constant size **p**) that is a value on each processor as opposed to the simple notation x which means the value of x on the current processor.

The transformation of BSP-*WhyML* codes into *WhyML* ones is based on the fact that, for each super-step, if we execute sequentially the code for each processor and then perform the simulation of the communications by copying the data, we have the same results as in really truly doing it in parallel.

The first step of the transformation is a decomposition of the program into blocks of sequential instructions — Fig. 3. Once that is done for each code block, we create a "for" loop to execute sequentially the block. That is the block is executed **p** times, once for each processor. Finally, we generate invariants to keep track of which variables are modified: since we are using arrays to represent the variables local to every processor and programs are run in a SPMD fashion, it is necessary to say that we only modify a variable on the current processor and that the rest of the array stays unchanged. Also, when transforming a `if` or `while` structure, there is a risk that a global synchronous instruction (a collective operation) might be executed on a processor and not on the other. We generate an assertion to forbid this case, ensuring that the condition associated with the instruction will always be true on every processor at the same time — thus forbidding deadlocks. The details and some examples are available in [9]. The trustworthiness of this tool is discussed in the conclusion.

3.3 BSP State-Space Construction

Algorithm "Normal" can be easily parallelised using a partition function cpu that returns for each state a processor id, *i.e.*, the processor numbered cpu(s) is the owner of s: **logic** cpu: state → int **axiom** cpu_range: ∀s:state. 0≤ cpu(s)<**nprocs**

```
let naive_state_space () =
 let total = ref 1 in
 let known = ref ∅ in
 let todo = ref ∅ in
 let pastsend = ref ∅ in
 if cpu(s0) = bsp_pid then
     todo←!todo ⊕ s0;
  while total>0 do
   let tosend = (local_successors
                     known todo pastsend) in
     exchange todo total !known
              pastsend !tosend
  done;
  !known
```

```
let local_successors (...) =
  let tosend = ref (init_send ∅) in
  while todo ≠ ∅ do
     let s = pick todo in
     known←!known ⊕ s;
     let new_states = ref ((succ s) \ !known \ !pastsend) in
     while new_states ≠ ∅ do
       let new_s = pick new_states in
       let tgt=cpu(new_s) in
         if tgt=bsp_pid
           then todo←!todo ⊕ new_s
           else tosend<tgt>←tosend<tgt> ⊕ new_s
       done
   done;
  !tosend
```

Fig. 4. Parallel (distributed) BSP-*WhyML* algorithm for state-space construction

The idea is that each process computes the successors for only the states it owns. This is rendered as the BSP algorithm of Fig. 4. Sets known and todo are still used but become local to each processor and thus provide only a partial view on the ongoing computation.

Function local_successors computes the successors of the states in todo where each computed state that is not owned by the local processor is recorded in a set tosend together with its owner number. The set pastsend contains all the states that have been sent during the past super-steps — the past exchanges. This prevents returning a state already sent by the processor: this feature is not necessary for correctness and consumes more memory but it is generally more efficient mostly when the state-space contains many cycles.

Function exchange is responsible for performing the actual communications: it returns the set of received states that are not yet known locally together with the new value of *total* — it is essentially the MPI's alltoall primitive.

To ensure termination of the algorithm, we use the additional variable *total* in which we count the total number of sent states. We have thus not used any complicated methods as the ones presented in [2]. It can be noted that the value of *total* may be greater than the intended count of states in *todo* sets. Indeed, it may happen that two processors compute a same state owned by a third processor, in which case two states are exchanged but then only one is kept upon reception. In the worst case, the termination requires one more super-step during which all the processors will process an empty *todo*, resulting in an empty exchange and thus $total = 0$ on every processor, yielding the termination.

3.4 Verification of the Parallel Algorithm

For lack of space, we only present the verification of the parallel part of this algorithm and not the sequential local_successors (similar to "Normal" but with many additional invariants on states to send) nor exchange — which is more technical and without really interesting properties and still available in the source code: the exchange procedure is only a permutation of the states that is, from a global point of view, only states in arrays have moved and there is no loss of states and a state has not magically appeared during the communications. Fig. 5 gives the annotated parallel algorithm. We also use the following predicates:

- isproc(i) is defined what is a valid processor's id that is $0 \leq$ i<**nprocs**;
- $\bigcup$(<p_set>) is the union of the sets of the **p**-value p_set that is $\bigcup_{i=0}^{\mathbf{p}}$p_set<i>;
- GoodPart(<p_set>) is used to indicate that each processor only contains the states it owns that is $\forall$i:int. isproc(i) $\rightarrow$ $\forall$s:state. s $\in$p_set<i> $\rightarrow$ cpu(s)=i;
- comm_send_i(s,j) is the set of sent states from processor i to processor j.

As before, we need to prove that (1) the code does not fail; (2) indeed computes the entire state-space and (3) terminates. The first property follows immediately since only the routine pick is used as before; and to also prove that the code is deadlock free (the loop contains exchange which implies a global synchronisation of all the processors), we can easily maintain that total (which gives the condition for termination) has the same value on all the processors during the entire execution of the algorithm. Let us now focus on the two other properties.

Correctness of the Parallel Loop (Fig. 5). The invariants (lines $9-18$) of the main parallel loop work as follows: (1) as in "Normal", we need to maintain that known (even distributed) is a subset of StSpace which finally ensures (A) when todo is empty; (2) as usual, the states to be treated are not already known; (3) our sets are well distributed (there is no duplicate state that is, each state is only kept in a unique processor); (4) total is a global variable, we thus ensure that it has the same value on each processor; (5) ensures that no state remains in todo (to be treated) when leaving the loop since total is at least as big as the cardinality of todo, total is an over-approximation of the number of sent states; (6–8), as usual, ensure property (B); (9) past sending states are in the state-space; (10) pastsend only contains states that are not owned by the processor and (11) all these states, that were sent, are finally received and stored by a processor.

In the post-condition (line 26), we can also ensures that the result is well distributed: the state-space is complete and each processor only contains the states it owns according to the function "cpu".

Termination (Fig. 5). For the local computations, the termination is ensured as in the "Normal" algorithm since known can only grow when entering the loop.

The main loop is more subtle: total is an over-approximation and thus could be greater to 0 whereas todo is empty. This happens when all the received states are already in known. The termination has thus two cases: (a) in general the set known globally (that is, from a global point of view, of all processors) grows and we have thus the cardinality of StSpace minus known which is strictly decreasing; (b) if there is no state in any todo of a processor (case of the last super-step), no new states would be computed and thus total would be equal to 0 in the last stage of the main loop.

We thus used a lexicographic order (this is well-founded ensuring termination) on the two values: sum of known across all processors; and total (which is the same on all processors) when no new states are computed and thus when no state

```
let naive_state_space () =
  let known = ref ∅      in let todo = ref ∅ in
  let pastsend = ref ∅   in let total = ref 1 in
    if cpu(s0) = bsp_pid then
     todo ←s0 ⊕ !todo;
    while total>0 do
     {
      invariant
     (1) ⋃(<known>) ∪ ⋃(<todo>) ⊆ StSpace
and (2) (⋃(<known>) ∩ ⋃(<todo>))=∅
and (3) GoodPart(<known>) and GoodPartt(<todo>)
and (4) (∀ i,j:int. isproc(i) → isproc(j) → total<i> = total<j>)
and (5) total<0> ≥ |⋃(<todo>)|
and (6) s0 ∈(⋃(<known>) ∪ ⋃(<todo>))
and (7) (∀ e:state. e ∈⋃(<known>) → succ(e) ⊆ (⋃(<known>) ∪ ⋃(<todo>)))
and (8) (∀ e:state. ∀i:int. isproc(i) → e ∈known<i> → succ(e) ⊆ (known<i> ∪ pastsend<i>))
and (9) ⋃(<pastsend>) ⊆ StSpace
and (10) (∀ i:int. isproc(i) → ∀e:state. e ∈pastsend<i> → cpu(e)≠ i)
and (11) ⋃(<pastsend>) ⊆ (⋃(<known>) ∪ ⋃(<todo>))
      variant pair(total<0>,| S \ ⋃(known) |) for lexico_order
     }
      let tosend=(local_successors known todo pastsend) in
       exchange todo total !known !tosend
    done;
    !known
    {StSpace=⋃(<result>) and GoodPart(<result>)}
```

Fig. 5. Parallel annotated algorithm

would be sent during the next super-step. At least, one processor cannot received any state during a super-step. We thus need an invariant in the local_successors for maintaining the fact that the set known potentially grows with at least the states of todo. We also maintain that if todo is empty then no state would be sent (in local_successors) and received, making total equal to 0 — in exchange.

Mechanical Proof. With some obvious axioms on the above predicates (such as ⋃<∅,...,∅>=∅) so that solvers can handle the predicates, all the produced obligations are automatically discharged by a combination of the solvers. In the following table, for each part of the parallel algorithm, we give the number of obligations and how many are discharged by the provers (some proof obligations require long timeouts *e.g.* 10 mins):

part/Solvers	Total	Alt-Ergo	Simplify	Z3	CVC3	Yices	Vampire
main	106	74	98	101	0	54	78
successor	46	16	42	41	24	14	32
exchange	24	20	22	23	0	16	15

Now the combination of all provers is needed since none of them is able to prove all the obligations. This is certainly due to their different heuristics. We also note that Simplify and Z3 remain the most efficient. Some obligations are hard to follow due to the parallel computations. But reading them carefully, we can find the good annotations. An interesting point is that the first author with the help of an undergraduate student was able to perform the job (annotate this parallel algorithm) in three months. Based on this fact, it seems conceivable that

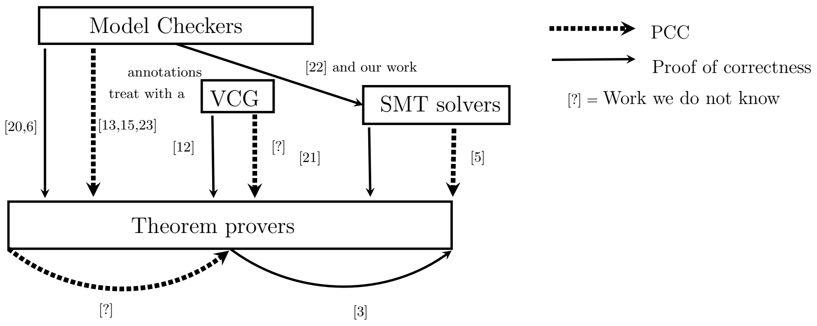


Fig. 6. Different ways for proving model-checking algorithms

a more seasoned team in formal methods can tackle more substantial algorithms (of model-checking) in a real programming language.

4 Related Work

Other Methods for Proving the Correctness of Model-Checkers. Fig. 6 summarises different methods that have been used for verifying MCs where each arrow corresponds to a proof of correctness (using a theorem prover or a PCC approach) and the papers related to the work.

The state-space explosion can be a problem for MCs extracted from theorem provers. They are pure functional programs such as the ones of [20,6]. They certainly would be too slow for big models even if there work on obtaining imperative programs from extracted (pure) functional programs.

The "certifying model-checking" is an established research field [15,23]. But, the performance issue of PCC is discussed in [26] and [16] where the authors present developments (and model-checking benchmarks) of BDDs and tree automata using theorem provers: BDDs are common data-structures used by MCs and tree automata is an approach for having a formal successor function. PCC only focuses on the generation of independently-checkable evidences as the compiled code satisfies a simple behavioural specification such as memory safety; the evidence can then be checked efficiently. Using PCC for state-space is the same as computing it a "second time". In fact, the drawback of proof certificates is that verification tools have to be instrumented to generate them, and the size of fully expanded proofs may be too large. Authors of [26,16] conclude that PCCs are here inadequate and we can conclude that MCs themselves need to be proved. It is also the conclusion of [8] where the authors note that "to avoid the inefficiency of fully expansive proof generations, a number of researchers have advocated the verification of decision procedures".

Using annotations in source codes (programs or algorithms) and a VCG has the advantage that realistic and efficient codes (mainly imperative ones) may be verified which could be difficult using theorem provers. And it will not be worth checking all the execution results of the MCs (which can take time) as in the PCC approach because the results will be guaranteed.

In our work, we also only use automatic solvers for proving the generated goals of the VCG *WHY* and thus we do not use any "elaborate" theorem prover such as Coq. The correctness of our results depends on the correctness of (1) the *WHY* tool (correct generation of goals) and (2) the results of the solvers. Relying on modules like SMT solvers has the advantage that these tools would certainly be verified in a close future. The work of [12] is a first approach for (1) and the work of [5] is a PCC approach for (2). Moreover, a SMT solver has been proved using a theorem prover [21]. In a close future, we can hope to achieve the same confidence in our codes as the MCs extracted from [20,6], as well as better performances since our codes are realistic imperative codes — and not functional ones from theorem provers. Finally, we think that using annotations (and a VCG tool) has the advantage of being "easy". And we can prove the correctness of programs or limit the work to some safety properties if the full correctness is too difficult to obtain. And it extends to parallel programs which is not easy using PCCs or theorem provers.

Other Various Works. There are also interesting examples of verified algorithms on *WHY*'s web page: Dijkstra shortest path, sorting, Knuth-Morris-Pratt string searching, *etc.* A mechanically assisted proof using Isabelle of how LTL formulae can be transformed into Büchi automata is presented in [17]. CTL* temporal logic is also available in Coq [24]. All these works are interesting since logical theories may be axiomatised in *WHY*.

Model compilation is one of the numerous techniques to speedup model-checking: it relies on generating source code (then compiled into machine code) to produce a high-performance implementation of the state-space exploration primitives, mainly the successor function. In [10], authors propose a way to prove the correctness of such an approach. More precisely, they focus on generated Low-Level Virtual Machine (LLVM) code from high-level Petri nets and manually prove that the object computed by its execution is a representation of the compiled model. If such a work can be redone using a theorem prover, we will have a machine-checked successor function which is currently axiomatised in *WHY*.

5 Conclusion

Model checkers are specialised software, using sophisticated algorithms, whose correctness is vital. In this work, we focus on correctness of well-known sequential algorithms for finite state-space construction (which is the basis for explicit model-checking) and on a distributed one designed by the authors. We annotated the algorithms for finite sets operations (available in Coq) and used the VCG *WHY* (certifying in Coq [12]) to obtain goals that were entirely checked by automatic solvers. These goals ensure the termination of the algorithms as well as their correctness for any successor function — assumed correct and generating a finite state-space. We thus gained more confidence in the code. We also hope to have convinced the reader that this approach is humanly feasible and applicable to real (parallel or sequential) model-checking algorithms.

Future goals are clear. First, adapt this work for true MC algorithms — as those for LTL/CTL* mostly Tarjan/NDFS like algorithms. This is challenging in general but using an appropriate VCG, we believe that a team can “quickly” do it. Second, we are currently proving algorithms and not real codes. Regarding the code structure, this is not really an issue and translating the resulting proof into a verification tool for true programs should be straightforward, mostly if high level data-structures are used: the *WHY* framework allows a user to generate WhyML code from Java using a tool call Krakatoa. Third, the successor function (computation of the transitions of the state-space) is currently an abstract function. We think to prove (mechanically) the work of [10] to compensate for this deficiency. Fourth, compressions aspects (symmetry, partial order, *etc.*) must be studied. The work of [25] which uses the B method could be a good basis. And to finish, the transformation of BSP-*WhyML* into *WhyML* is potentially not correct. The second author is working on this. The effort for all these works and thus verifying the whole stack of Fig. 6 is not at all within the reach of a single team. But our guess is that each of these stages is largely feasible. Also, machine-checked MCs would certainly be less efficient than traditional ones. But they could be used in addition when it comes to giving greater confidence in the results. We also believe that another interesting application of a verified tool (such as we are envisioning) would be to serve as a reference implementation that is used to compare the results of an efficient implementation over a set of benchmark problems.

References

1. Armand, M., Grégoire, B., Spiwack, A., Théry, L.: Extending Coq with imperative features and its application to SAT verification. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 83–98. Springer, Heidelberg (2010)
2. Barnat, J.: Distributed Memory LTL Model Checking. PhD thesis, Faculty of Informatics Masaryk University Brno (2004)
3. Barras, B., Werner, B.: Coq in Coq. Technical report, INRIA (1997)
4. Bisseling, R.H.: Parallel scientific computation. A structured approach using BSP and MPI. Oxford University Press (2004)
5. Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 179–194. Springer, Heidelberg (2010)
6. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.-G.: A fully verified executable LTL model checker. In: Computer Aided Verification, CAV (to appear, 2013)
7. Filliâtre, J.-C.: Verifying two lines of C with why3: An exercise in program verification. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 83–97. Springer, Heidelberg (2012)
8. Ford, J., Shankar, N.: Formal verification of a combination decision procedure. In: Voronkov, A. (ed.) CADE 2002. LNCS (LNAI), vol. 2392, pp. 347–362. Springer, Heidelberg (2002)
9. Fortin, J., Gava, F.: BSP-WHY: an intermediate language for deductive verification of BSP programs. In: High-Level Parallel Programming and Applications (HLPP), pp. 35–44. ACM (2010)

10. Fronc, L., Pommereau, F.: Towards a certified Petri net model-checker. In: Yang, H. (ed.) APLAS 2011. LNCS, vol. 7078, pp. 322–336. Springer, Heidelberg (2011)
11. Garavel, H., Mateescu, R., Smarandache, I.M.: Parallel state space construction for model-checking. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 217–234. Springer, Heidelberg (2001)
12. Herms, P.: Certification of a chain for deductive program verification. In: Bertot, Y. (ed.) 2nd Coq Workshop, Satellite of ITP 2010 (2010)
13. Namjoshi, K.S.: Certifying model checkers. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 2–13. Springer, Heidelberg (2001)
14. Necula, G.C.: Proof-carrying code. In: Principles of Programming Languages (POPL), pp. 106–119. ACM (1997)
15. Peled, D., Pnueli, A., Zuck, L.D.: From falsification to verification. In: Hariharan, R., Mukund, M., Vinay, V. (eds.) FSTTCS 2001. LNCS, vol. 2245, pp. 292–304. Springer, Heidelberg (2001)
16. Rival, X., Goubault-Larrecq, J.: Experiments with finite tree automata in Coq. In: Boulton, R.J., Jackson, P.B. (eds.) TPHOLs 2001. LNCS, vol. 2152, pp. 362–377. Springer, Heidelberg (2001)
17. Schimpf, A., Merz, S., Smaus, J.-G.: Construction of Büchi Automata for LTL Model Checking Verified in Isabelle/HOL. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 424–439. Springer, Heidelberg (2009)
18. Shankar, N.: Trust and automation in verification tools. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 4–17. Springer, Heidelberg (2008)
19. Skillicorn, D.B., Hill, J.M.D., McColl, W.F.: Questions and answers about BSP. Scientific Programming 6(3), 249–274 (1997)
20. Sprenger, C.: A verified model checker for the modal μ-calculus in coq. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 167–183. Springer, Heidelberg (1998)
21. Stump, A., Oe, D., Reynolds, A., Hadarean, L., Tinelli, C.: SMT proof checking using a logical framework. Formal Methods in System Design 42(1), 91–118 (2013)
22. Sun, J., Liu, Y., Cheng, B.: Model checking a model checker: A code contract combined approach. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 518–533. Springer, Heidelberg (2010)
23. Tan, L., Cleaveland, W.R.: Evidence-based model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 455–470. Springer, Heidelberg (2002)
24. Tsai, M.-H., Wang, B.-Y.: Formalization of cTL* in calculus of inductive constructions. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 316–330. Springer, Heidelberg (2008)
25. Turner, E., Butler, M., Leuschel, M.: A refinement-based correctness proof of symmetry reduced model checking. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 231–244. Springer, Heidelberg (2010)
26. Verma, K.N., Goubault-Larrecq, J., Prasad, S., Arun-Kumar, S.: Reflecting BDDs in Coq. In: Kleinberg, R.D., Sato, M. (eds.) ASIAN 2000. LNCS, vol. 1961, pp. 162–181. Springer, Heidelberg (2000)

Inductive Verification of Hybrid Automata with Strongest Postcondition Calculus

Daisuke Ishii[1], Guillaume Melquiond[2], and Shin Nakajima[1]

[1] National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, Japan
dsksh@acm.org, nkjm@nii.ac.jp
[2] INRIA Saclay–Île-de-France, LRI, bât 650, Université Paris Sud 11, Orsay, France
guillaume.melquiond@inria.fr

Abstract. Safety verification of hybrid systems is a key technique in developing embedded systems that have a strong coupling with the physical environment. We propose an automated logical analytic method for verifying a class of hybrid automata. The problems are more general than those solved by the existing model checkers: our method can verify models with symbolic parameters and nonlinear equations as well. First, we encode the execution trace of a hybrid automaton as an imperative program. Its safety property is then translated into proof obligations by strongest postcondition calculus. Finally, these logic formulas are discharged by state-of-the-art arithmetic solvers (e.g., Mathematica). Our proposed algorithm efficiently performs inductive reasoning by unrolling the execution for some steps and generating loop invariants from verification failures. Our experimental results along with examples taken from the literature show that the proposed approach is feasible.

1 Introduction

Hybrid systems, transition systems with continuous dynamics, are a good model for embedded systems that have a strong coupling with the physical environment. Achieving the desired reliability levels of such systems has brought a challenging and important problem in formal methods research.

To date, verification of hybrid systems has been extensively studied with two prominent approaches: *model checking* and *logical analysis*. The model-checking approach has been successfully applied to practical examples with tools such as HyTech [12], PHAVer [8], and HybridSAL [22]. The approach is said algorithmic: tools numerically over-approximate a certain class of *hybrid automata* (HA) to have piecewise-linear systems, and apply model-checking methods [4]. The second approach is based on logical analysis [16]. While the theory of logical analysis has been studied extensively, there are few practical tools. A notable and successful exception is KeYmaera [18]. The logical analytic approach can be applied to the class of *hybrid programs* which generalize the automata handled by model checking. Indeed, this class includes systems with symbolic parameters and nonlinear dynamics. There is, however, a major drawback: the larger the class of systems is, the less automatic its verification becomes. Engineers thus have to apply some proof strategies during the interactive verification process, which requires understanding the target model.

In this paper, we propose a partly automated tool for the logical analysis of HA that makes heavy use of state-of-the-art arithmetic solvers. Our goal is to prove safety properties. First, our method encodes executions of HA into straight-line imperative programs. This formalism allows us to construct a *lasso-shaped* structure based on induction: after exhibiting at most m steps of continuous evolution and discrete transition, any execution of the system forms a loop with a length of at most n steps between some specific regions of the state space. Then, the imperative program is transformed into a conjunction of verification conditions as a result of *strongest postcondition* (SP) calculus. The resulting logic formula involves real-arithmetic predicates and ordinary differential equations (ODEs). The generated conditions can be discharged using solvers such as Mathematica for some nonlinear HA.

The contribution of this work is as follows. The use of an imperative language and SP calculus gives a straightforward justification of the soundness of our method for generating the finite-length verification conditions from HA. The algorithm we propose realizes an automated verification process, although some user interactions are needed to determine efficiently (a) correct numbers m and n of steps to unroll the execution and (b) the loop invariant that represents the initial region of the loop. Computer algebra techniques, however, are employed to automate most of the work in generating loop invariants.

This paper is organized as follows. Section 2 introduces the class of hybrid automata. Section 3 describes a simple imperative language for simulating HA and the corresponding SP calculus. In Section 4, we present the concept of induction and loop unrolling, and describe an algorithm for automated verification. Section 5 describes an implementation using Mathematica. Section 6 reports how our implementation behaves on several examples and provides a comparison of the results with existing tools. Section 7 describes some related studies.

2 Hybrid Automata

In this paper, we model hybrid systems as hybrid automata (HA) [11].

Definition 1. *A* hybrid automaton *is a tuple $HA = \langle L, V, Init, \mathcal{G}, \mathcal{R}, \mathcal{F}, \mathcal{I} \rangle$ that consists of the following components:*

- *A finite set $L = \{l_1, \dots, l_p\}$ of* locations.
- *A finite set $V = \{x_1, \dots, x_q\}$ of real-valued* variables. *$\mathbb{R}^V$ is the set of all of the* valuations *of the system.*
- *An* initial condition *$Init$ in $L \times \mathbb{R}^V$ that specifies the* initial states.
- *A family $\mathcal{G} = \{G_{l,l'}\}_{l \in L, l' \in L}$ of* guard conditions *$G_{l,l'}$ in $\mathbb{R}^V$.*
- *A family $\mathcal{R} = \{R_{l,l'}\}_{l \in L, l' \in L}$ of* reset functions *$R_{l,l'} : \mathbb{R}^V \to \mathbb{R}^V$.*
- *A family $\mathcal{F} = \{F_l\}_{l \in L}$ of* vector fields *$F_l : \mathbb{R}^V \to \mathbb{R}^V$.*
- *A family $\mathcal{I} = \{I_l\}_{l \in L}$ of* location invariants *I_l in $\mathbb{R}^V$.*

$$\frac{t > 0 \quad \phi(0) = \nu \quad \forall \tilde{t} \in [0, t]\ \frac{d\phi}{dt} = F_l \wedge I_l[\phi(\tilde{t})]}{\langle l, \nu \rangle \xrightarrow{t} \langle l, \phi(t) \rangle} \qquad \frac{G_{l_1,l_2}[\nu_1] \quad \nu_2 = R_{l_1,l_2}(\nu_1) \quad I_{l_2}[\nu_2]}{\langle l_1, \nu_1 \rangle \xrightarrow{0} \langle l_2, \nu_2 \rangle}$$

Fig. 1. Operational semantics of HA

A (finite or infinite) execution *of HA is a sequence* $\sigma_0 \xrightarrow{t_1} \sigma_1 \xrightarrow{t_2} \cdots$*, for which* $\sigma_i \in L \times \mathbb{R}^V$ *and* $Init[\sigma_0]$ *holds, and* $\xrightarrow{*}$ *is either a* continuous evolution phase $\xrightarrow{t}$ *where* $t > 0$ *or a* discrete transition phase $\xrightarrow{0}$ *and is given by the rules in Figure 1. In the first rule,* $\frac{d\phi}{dt} = F_l$ *is an abbreviation of* $\frac{d\phi(\tilde{t})}{dt} = F_l(\phi(\tilde{t}))$*. We say that an execution is length-k* canonical *when of the form* $\sigma_0 \xrightarrow{t_1} \sigma_1 \xrightarrow{0} \sigma_2 \xrightarrow{t_3} \cdots \xrightarrow{0} \sigma_{2k}$ *that alternates continuous and discrete phases.*

In this paper, we assume that multiple discrete transitions do not occur in an instant. We also assume that no discrete transition occurs initially. Thus, any execution can be expressed as a canonical execution. Verification of non-canonical executions can be considered as future work. Infinite-length canonical executions are supported; yet in presence of Zeno points (infinite number of transitions in finite time), HA executions are handled only up to the first point.

Example 1. Water-level monitor (WLM) [3,15]. A controlled water tank is modeled as a four-location constant-rate HA, as illustrated in Figure 2. It supplies water at a constant rate $rate_{out}$, whereby, in location *off* (and *sw-on*), the water level y decreases as $\frac{d\phi_y}{dt} = rate_{out}$. In location *on* (and *sw-off*), the system pumps water to refill the tank, which results in the water level changing as $\frac{d\phi_y}{dt} = rate_{in}$. A sensor observes y and switches between the locations *on* and *off* when the level reaches the thresholds *low* or *high*. However, it takes *delay* seconds for switching, hence the locations *sw-on* and *sw-off*. In this paper, we constrain the values for the constant parameters as follows:

$$\begin{aligned} min \le low \wedge high \le max \wedge low < high \wedge delay > 0 \wedge \\ max \ge high + rate_{in} \cdot delay \wedge min \le low + rate_{out} \cdot delay. \end{aligned} \tag{1}$$

Because the discrete transition edges in the automaton form a single cycle, the trace of locations that were reached is the same for all of the executions.

Definition 2. *A* safety property *(or an* inductive invariance*) is expressed by a formula* $\Box P$*, where* P *is a predicate on* $L \times \mathbb{R}^V$*.* $HA \models \Box P$ *denotes that HA satisfies* $\Box P$*, that is, predicate* P *holds initially and is preserved by every discrete transition and continuous evolution.*

Example 2. In the following sections, we will prove that the level stays between a lower and an upper limit, which is expressed by the following safety property:

$$\Box(min \le y \le max).$$

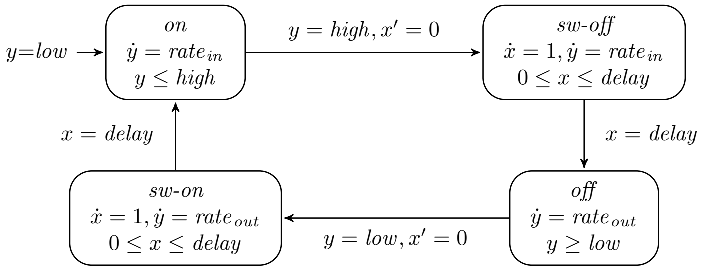


Fig. 2. Water-level monitor

3 Modeling HA Executions with Programs

In this section, we introduce the theoretical foundation of our method. It analyzes finite and infinite executions of a HA by reusing traditional tools in program verification. We first introduce a simple imperative language in which the statements simply sketch the executions of the HA (Section 3.1). Then, we provide a notion of strongest postcondition for each program statement given a precondition, and we prove that this calculus derives the safety of the HA (Section 3.2).

3.1 Imperative Language

Given a *HA*, we define an untyped imperative language $\mathbf{Imp}_{HA}$. This language is basic, since it does not even provide loops. For the purpose of this work, it has only sequences and the commands `evolve` and `trans`. The command `evolve` expresses a continuous evolution of the *HA* for a given duration, while `trans` expresses a discrete transition.

Definition 3. *The language* $\mathbf{Imp}_{HA}$ *is given by the following syntax:*

$$s ::= \texttt{skip} \mid s;\;\; s \mid \texttt{evolve}\ t \mid \texttt{trans}$$

Definition 4. *A* program state *(denoted S or S_i) is a map from variable names to program values. A special variable x_s is associated to the "current" state ($\in L \times \mathbb{R}^V$) of the HA execution. For the sake of readability, pseudo-variables are introduced to access part of the HA state as follows: $x_s = \langle x_l, \cdot \rangle = \langle \cdot, x_v \rangle = \langle \cdot, (x_1, \ldots, x_i, \ldots, x_q) \rangle$. We assume this equivalence is always maintained automatically when a new value is assigned to a pseudo-variable.*

Figure 3 describes the operational semantics of the language. $[\![e]\!]_S$ denotes the term obtained by replacing each free variable of an expression e by its associated value in the program state S. $S\{x \mapsto v\}$ denotes the program state obtained by adding to S that variable x is associated to the value v. The rules for `skip` and sequence are the usual ones. The rules for `evolve` and `trans` are derived from the operational semantics of a HA execution. Note that we allow the statement `evolve` 0, so that the theorems presented in this paper have a simple way to check the safety property for the initial state or after a discrete transition.

$$\frac{}{S, (\texttt{skip};\ s) \rightsquigarrow S, s} \qquad \frac{S_1, s_1 \rightsquigarrow S_2, s_2}{S_1, (s_1;\ s_3) \rightsquigarrow S_2, (s_2;\ s_3)} \qquad \frac{}{S, \texttt{evolve}\ 0 \rightsquigarrow S, \texttt{skip}}$$

$$\frac{[\![x_s]\!]_S \xrightarrow{t} \sigma}{S, \texttt{evolve}\ t \rightsquigarrow S\{x_s \mapsto \sigma\}, \texttt{skip}} \qquad \frac{[\![x_s]\!]_S \xrightarrow{0} \sigma}{S, \texttt{trans} \rightsquigarrow S\{x_s \mapsto \sigma\}, \texttt{skip}}$$

Fig. 3. Operational semantics of $\mathbf{Imp}_{HA}$

Lemma 1. *For any execution $\sigma_0 \xrightarrow{t_1} \sigma_1 \xrightarrow{0} \cdots \xrightarrow{t_k} \sigma_{2k-1}$ of the HA, assuming that $\sigma = \sigma_0$ holds for the initial program state, there is an execution of the following $\mathbf{Imp}_{HA}$ program that does not block (that is, it reduces to* `skip`*) and such that the final program state satisfies $x_s = \sigma_{2k-1}$.*

$$\texttt{evolve}\ t_1;\ \texttt{trans};\ \cdots;\ \texttt{evolve}\ t_k$$

Note that this program might also have either blocking executions or executions that end on a different HA state; the former are made irrelevant by our SP-based approach, while the latter are expected due to the non-deterministic nature of HA. For the programs above, the execution is canonical only for the first $k-1$ continuous steps; the last duration t_k can be arbitrarily short. It can also be arbitrarily large, if the HA stays infinitely long in that continuous evolution.

Since we can now express any partial execution of a HA as a program, we can state the safety property of the HA as a property that every non-blocking program must satisfy in its final state.

Lemma 2. *If, for all non-blocking programs of $\mathbf{Imp}_{HA}$ of the above form starting from an initial program state $\sigma \in Init$, property P holds in the final program state, then P is a safety property for the HA (up to the first Zeno point, if any).*

3.2 Strongest Postconditions

In this section, we instantiate the principles of program verification [13,7] with $\mathbf{Imp}_{HA}$. We are not interested in manual verification, so we will skip over the definition of Hoare triples and directly go to the topic of verification conditions (VCs). Moreover, since we are not dealing with reachability but only safety, we do not have to prove that programs are non-blocking, we can just assume they are. Therefore, weakest preconditions (WPs) and strongest postconditions (SPs) are dual from each other for our purpose. Should we have to perform backward reachability analysis, WP computation would be better suited. This is not the case though, so we choose SP, so as to follow the direction of time.

Lemma 3 (Soundness of SP). *For any program s in $\mathbf{Imp}_{HA}$, if the initial state satisfies a given property P, the final state satisfies $SP(P, s)$ (assuming s terminates) with SP inductively defined as follows.*[1]

[1] $P[x \leftarrow e]$ denotes the substitution of all the occurrences of variable x in P with e.

$$SP(P, \texttt{skip}) := P \qquad SP(P, s_1;\ s_2) := SP(SP(P, s_1), s_2)$$

$$SP(P, \texttt{evolve}\ t) := \exists\phi\ P[x_v \leftarrow \phi(0)] \wedge \phi(t) = x_v \wedge (\forall\tilde{t} \in [0, t]\ \tfrac{d\phi}{dt} = F_{x_l} \wedge I_{x_l}[\phi(\tilde{t})])$$

$$SP(P, \texttt{trans}) := \exists\langle l', x_v'\rangle\ P[x_s \leftarrow \langle l', x_v'\rangle] \wedge G_{l',x_l}[x_v'] \wedge x_v = R_{l',x_l}(x_v') \wedge I_{x_l}[x_v]$$

Proof. Let us assume that there are S and S' such that $[\![P]\!]_S$ holds and $S, s \rightsquigarrow^* S', \texttt{skip}$. We just have to prove that $[\![SP(P, s)]\!]_{S'}$ holds. The proof is performed inductively on the structure of the statement s by checking that every case of SP is implied by the operational semantics of $\mathbf{Imp}_{HA}$. This is a consequence of the operational semantics of HA given on Figure 1. □

Example 3. Let us prove that, if the HA of Figure 2 is in a state satisfying $x_l = on \wedge y = low$, then any continuous evolution of duration t leads to a state satisfying $y \leq max$. By Lemmas 2 and 3, it is sufficient to prove that the following implication holds in any program state:

$$SP((x_l = on \wedge y = low), \texttt{evolve}\ t) \Rightarrow y \leq max.$$

Let us assume that we are in program state such the left-hand side holds, and we prove that $y \leq max$ holds. From the definition of SP, we know that there exists a function ϕ such that

$$(x_l = on \wedge \phi_y(0) = low) \wedge \phi(t) = (x, y) \wedge (\forall\tilde{t} \in [0, t]\ \tfrac{d\phi}{dt} = F_{x_l} \wedge I_{x_l}[\phi(\tilde{t})])$$

As a consequence, we have $y = low + rate_{in} \cdot t$ (by solving the ODE) and $y \leq high$ (by unfolding the location invariant I_{x_l}). The latter property, in conjunction with Constraint (1) of Example 1, proves the goal $y \leq max$ by linear arithmetic.

Remark 1. As we will later pass the verification conditions to automated tools, it is important to eliminate as many quantifiers as possible beforehand. For instance, $SP(P, \texttt{trans})$ has the form $\exists l'\ Q[l']$. This is equivalent to the disjunction $Q[l_1] \vee \ldots \vee Q[l_p]$ with $l_1, \ldots, l_p$ all the locations. In the case of $SP(P, \texttt{evolve}\ t)$, Example 3 shows how one can get rid of $\exists\phi$ if the ODE admits a closed form.

4 Inductive Verification Method

4.1 Induction Strategy

We now present an algorithm derived from Lemma 2 that performs safety verification of a HA. The statement of Lemma 2 is unpractical, as it requires verifying infinitely-many programs. This section describes how we can build weaker yet more practical variants of it, by only considering a bounded number of programs. The approach is as follows. Let us assume that there is a predicate P^+ such that $P^+ \Rightarrow P$ and

– from an initial state, any execution of HA reaches a state satisfying P^+ after alternating at most m continuous evolutions and m discrete transitions,

– from any state satisfying P^+, any execution of *HA* reaches a state satisfying P^+ after alternating at most n continuous evolutions and n discrete transitions.

Verifying the safety property is therefore simple:

– For the initial m-step execution, we check that every intermediate state is safe and that the execution finally reaches the region represented by predicate P^+ (base case).
– For the n-step execution from the region P^+, we check that every intermediate state is safe and that the execution finally reaches the region P^+.

The success of our approach depends on whether we can exhibit some lengths m and n and some predicate P^+ for a given HA.

We first show the simplest case ($m = 0$ and $n = 1$): the base case is the verification of the initial states, and the induction is performed on a continuous phase followed by a discrete phase.

Theorem 1 (Simplest case). *Given a predicate P^+ such that $P^+ \Rightarrow P$ holds in any state, the following inference rule is correct:*

$$\frac{VC_0 : \mathit{Init} \Rightarrow P^+ \qquad \begin{array}{l} VC_1 : \forall t \geq 0 \; SP(P^+, \texttt{evolve } t) \Rightarrow P \\ VC_{-1} : \forall t \geq 0 \; SP(P^+, \texttt{evolve } t; \texttt{trans}) \Rightarrow P^+ \end{array}}{HA \models \Box P}$$

Proof. VC_0 checks that the initial states satisfy the property P^+. VC_{-1} inductively verifies that all of the possible two consecutive continuous and discrete phases $\sigma_i \xrightarrow{t_{i+1}} \sigma_{i+1} \xrightarrow{0} \sigma_{i+2}$ evolve for the arbitrary duration t_{i+1} from a state σ_i that satisfies P^+ to a state σ_{i+2} that again satisfies P^+. VC_1 ensures that the safety property was not broken during the continuous phase. $\Box$

We now extend the above induction to a more generic case.

Theorem 2 (Unrolled case).

$$\frac{\begin{array}{lll} SP_1 \equiv SP(\mathit{Init} \wedge \neg P^+, \texttt{evolve } t_1) & VC_1 & : \forall t_1 \geq 0 \; SP_1 \Rightarrow P \\ SP_2 \equiv SP(SP(SP_1, \texttt{trans}) \wedge \neg P^+, \texttt{evolve } t_2) & & \\ & VC_2 & : \forall t_1, t_2 \geq 0 \; SP_2 \Rightarrow P \\ & \vdots & \\ SP_m \equiv SP(SP(SP_{m-1}, \texttt{trans}) \wedge \neg P^+, \texttt{evolve } t_m) & & \\ & VC_m & : \forall t_1 \ldots t_m \geq 0 \; SP_m \Rightarrow P \\ SP_0 \equiv SP(SP_m, \texttt{trans}) & VC_0 & : \forall t_1 \ldots t_m \geq 0 \; SP_0 \Rightarrow P^+ \\ SP_{m+1} \equiv SP(P^+, \texttt{evolve } t_1) & VC_{m+1} & : \forall t_1 \geq 0 \; SP_{m+1} \Rightarrow P \\ & \vdots & \\ SP_{m+n} \equiv SP(SP(SP_{m+n-1}, \texttt{trans}) \wedge \neg P^+, \texttt{evolve } t_n) & & \\ & VC_{m+n} & : \forall t_1 \ldots t_n \geq 0 \; SP_{m+n} \Rightarrow P \\ SP_{-1} \equiv SP(SP_{m+n}, \texttt{trans}) & VC_{-1} & : \forall t_1 \ldots t_n \geq 0 \; SP_{-1} \Rightarrow P^+ \end{array}}{HA \models \Box P}$$

Proof. This theorem is an extension of Theorem 1. It verifies that a state satisfying P^+ can be reached in at most m steps initially (from VC_1 to VC_0), and then inductively that P^+ can always be reached again in at most n steps (from VC_{m+1} to VC_{-1}). $\square$

Remark 2. Only VC_0 and VC_{-1} check that P^+ holds after an execution; all the other VCs check the safety property P only. Moreover, except for VC_{m+1}, all these other conditions compute the SP by assuming that P^+ does not hold. Indeed, there might be less than n transitions before reaching again a state satisfying P^+ (or m transitions initially).

4.2 Verification Algorithm

Given a HA, a safety property $\Box P$, and the maximal numbers m_{max} and n_{max} of steps to unroll, the algorithm in Figure 4 tries to check that all the hypotheses of Theorem 2 hold, and thus that $HA \models \Box P$ holds too.[2] The algorithm performs the inductive verification with every $m \leq m_{max}$ and $n \leq n_{max}$ (line 1). It iteratively computes the base case (line 4) and then the induction step (line 7). Procedure Validate returns *true* if the given logic formula holds, *false* if it cannot conclude. The verification succeeds if all the verification conditions are successfully validated (line 10).

When the verification fails during the induction step, we strengthen the loop invariant (line 8) so that the failing condition holds, and we perform the verification anew. Possibly, procedure Learn strengthened the invariant so much that we detect it is now useless (line 3). In this case, we leave from the inner recursion and try the verification with another m and n, or otherwise return *false*.

Note that the algorithm does not specify how to enumerate m and n. Typically, we enumerate from $m = 0$ and $n = 1$, but for certain models, we can guess the values, e.g., from the size of a lasso-shaped automaton. In the algorithm, the verification of the base case (lines 4-6, named BaseCase) and the induction step (lines 7-11, named Induction) are independent, thus we can also reverse the order of the two verification processes.

4.3 Loop Invariant Generation

In the following, we present the loop invariant generation method implemented in procedure Learn. Let us assume that the verification of a condition $VC_i \equiv \forall t_1 \dots t_i \geq 0 \; SP(P^+, s) \Rightarrow P$ has failed in the induction step. Then, Learn(VC_i) generates a lemma Q from the failed verification. Specifically, Learn generates a formula Q such that VC_i becomes valid after we update the loop invariant as $P^+ := P^+ \wedge Q$. Basically, Learn searches for Q such that $SP(Q, s) \Rightarrow VC_i$ holds by applying algebraic transformations to VC_i. Note that all the occurrences of variable x_s (the current state) in Q refer to the time P^+ holds, while the ones in VC_i refer to the state at the end of the execution of s. To fix this discrepancy,

[2] A failure of the algorithm does not imply that the safety property is invalid.

```
Input: $HA$; $P$; $m_{max} \in \mathbb{N}_{\geq 0}$; $n_{max} \in \mathbb{N}_{>0}$
Output: $true$: $HA \models \Box P$; $false$: cannot decide $\Box P$ within $m_{max} + n_{max}$ steps
 1: for $m \in \{0, \ldots, m_{max}\}$; $n \in \{1, \cdots, n_{max}\}$ do
 2:    $P^+ := P$
 3:    while $P^+ \not\equiv false$ do
 4:       if $\neg\forall i \in \{0, \ldots, m\}$ Validate($VC_i$) then
 5:          break
 6:       end if
 7:       if $\exists j \in \{m+1, \ldots, m+n, -1\}$ $\neg$Validate($VC_j$) then
 8:          $P^+ := P^+ \wedge$ Learn($VC_j$)
 9:       else
10:          return $true$
11:       end if
12:    end while
13: end for
14: return $false$
```

Fig. 4. Algorithm for inductive verification

Learn computes Q by using a *quantifier elimination* (QE) method, such as the Resolve procedure of Mathematica:

$$Q := \mathsf{QE}(\forall x_s \forall t_1 \ldots t_i \ (SP((P^+ \wedge x_0 = x_s), s) \Rightarrow P))[x_0 \leftarrow x_s].$$

To simplify the loop invariant, the other local variables in VC_i, i.e., $\phi, \tilde{t}, l', x'_v$ introduced in the SP calculus in Lemma 3 and t_i introduced in Theorem 2, should also be removed. Unfortunately, QE with mixed quantifiers and function quantifiers is a hard problem in general. See Remark 1 and the next section for details on how we perform this simplification.

The formula computed for Q is often a large disjunctive formula that is unusable as a loop invariant. For instance, some sub-formulas of Q describe states that are never accepted by the HA. Such sub-formulas are not only useless but make the verification process expensive. So we strengthen Q according to the following strategies:

– *Lemma separation.* We split Q at the (top-most) disjunction operators and employ one (or several) of the resulting sub-formulas.
– *Location disabling.* When we remove a sub-formula of Q that is related to some location l, we insert the constraint $x_l \neq l$. The resulting loop invariant might be effective when combined with loop unrolling.

5 Implementation

We have implemented the method presented in the previous sections using Mathematica 8.0.4[3], which can perform the computations in a fully symbolic manner.

[3] http://www.wolfram.com/mathematica/

Note that the loop invariant generation by Learn (line 8) is not automatic but guided by the user so as to apply the strategies described in Section 4.3. Validate is implemented in three different ways by using the built-in procedures of Mathematica, FullSimplify, Reduce, and FindInstance. We also rely on Mathematica's DSolve to find closed form of ODEs whenever possible.

In the implementation of BaseCase and Induction, we optimize the computation in two ways. First, we do not validate each VC_i separately but try to reuse the common assumptions. When validating VC_i, the algorithm computes SP_i which axiomatizes the state after executing the corresponding program s_i, and then validates $SP_i \Rightarrow P/P^+$. If we perform the validation of VCs in ascending order, we can compute SP_i from SP_{i-1} efficiently. Second, we perform location-wise validation of VCs to avoid the inefficiency that occurs when the execution of program s spans multiple locations. So we replicate the SP and instantiate each copy with a different location (cf. Remark 1). Throughout the computation, we manage the set of the copies instead of the original SP. Although it causes Validate to be called more often, the computation is more efficient in general.

Example 4. We verify the safety property of Example 2 for the HA in Example 1 with this implementation. Following the main algorithm, we first compute with $m = 0$ and $n = 1$. We run BaseCase to check that *Init* entails $P^+ \equiv P$, and it returns *true*. Next, we simulate a continuous and discrete change by running Induction. It computes the SP separately for each of the locations, *on*, *sw-off*, *off*, and *sw-on*, and validates VCs. For VC_1, the validation for locations *on* and *off* succeeds but the validation for *sw-off* and *sw-on* fails. Procedure Learn generates the following lemmas for these two locations.

$$\begin{aligned}
Q_{\textit{sw-off}} \equiv\ & \textit{min} + x \cdot \textit{rate}_{\textit{in}} \le y + \textit{delay} \cdot \textit{rate}_{\textit{in}} \le \textit{max} + x \cdot \textit{rate}_{\textit{in}} \vee \\
& x = \textit{delay} \ \vee\ y + \textit{delay} \cdot \textit{rate}_{\textit{in}} < \textit{low} + x \cdot \textit{rate}_{\textit{in}}, \\
Q_{\textit{sw-on}} \equiv\ & \textit{min} + x \cdot \textit{rate}_{\textit{out}} \le y + \textit{delay} \cdot \textit{rate}_{\textit{out}} \le \textit{max} + x \cdot \textit{rate}_{\textit{out}} \vee \\
& x = \textit{delay} \ \vee\ \textit{high} + x \cdot \textit{rate}_{\textit{out}} < y + \textit{delay} \cdot \textit{rate}_{\textit{out}}.
\end{aligned}$$

Here, we can use either of the two presented strategies for improving the loop invariant. For instance, location disabling appends

$$Q_1 := x_l \ne \textit{sw-off} \wedge x_l \ne \textit{sw-on}$$

to P^+. The VCs are then successfully validated with $m = 0$ and $n = 2$.

The lemma-separation strategy makes use of the additional lemmas generated by Learn. Here, we divide each lemma into three parts at the top-most disjunction operator. Then, the first part of each lemma (denoted $Q_{\textit{sw-off},1}$ and $Q_{\textit{sw-on},1}$) makes the verification successful. More precisely, if we append

$$Q_2 := (x_l = \textit{sw-off} \Rightarrow Q_{\textit{sw-off},1}) \wedge (x_l = \textit{sw-on} \Rightarrow Q_{\textit{sw-on},1})$$

to P^+, the validation succeeds with $m = 0$ and $n = 1$.

6 Experiments

To confirm the feasibility of our method and to compare it with existing tools, we applied it to several examples taken from the literature. We also verified the

Table 1. Experimental results.

example	locs	vars	unroll	lemmas	Mathematica	MC tool	KeYmaera
WLM (Ex. 1)	4	2	0/1	2	0.85s	–	1.8s
LGB	2	3	4/2	3	2.22s	0.004s (H)	–
temp. control	4	3	1/1	4	2.82s	0.012s (H)	–
bouncing ball	1	2	0/1	1	0.49s	–	0.9s
ETCS	2	3	0/1	1	4.48s	–	3.1s
highway 9	10	9	0/2	1	0.22s	0.22s (P)	–
highway 19	20	19	0/2	1	3.64s	–	–

examples using the existing tools, HyTech, PHAVer, and KeYmaera, for comparison. The encoded models for the implementation is available at `http://www.ueda.info.waseda.ac.jp/~ishii/pub/mathybrid/`. Table 1 reports the results of verifying the examples using our implementations. The columns are: the number of locations; the number of variables; the way loops are unrolled (i.e., m/n); how many times P^{+} had to be improved by the main algorithm; the computational time taken by the BaseCase and Induction procedures implemented in Mathematica; the time taken by HyTech (version 1.04f, indicated by "H") or PHAVer (version 0.38, indicated by "P"); and the time taken by KeYmaera (version 3.0). The notation "–" means that the verification failed. The experiments were run on a 3.4GHz Intel Xeon processor with 4GB of RAM. Note that the computational time for our method only measures the process after we found the loop invariants, since their generation requires some human interaction.

6.1 Considered Examples

WLM. Example 1 could be verified with our proposed method in a reasonable time, as explained in Example 4. In [15], the same instance was handled by using a mathematical solver manually, whereas our Mathematica implementation verified the instance by simply following the algorithm. The model-checking (MC) tools could not handle this instance because of the nonlinear terms caused by the parameterized flow rate. KeYmaera verified this example but the model had to be given a loop invariant beforehand [17].

Leaking gas burner (LGB) [3]. Our implementation verified this rectangular HA consisting of two locations $L = \{leaking, non\text{-}leaking\}$ as follows: Induction failed in the verification of the first continuous evolution in the two locations. The lemma generated for *leaking* was successful. For *non-leaking* though, we had to resort to our location-disabling strategy. Then, the verification succeeded with $m = 4$ and $n = 2$. This model was verified efficiently by the MC tools. KeYmaera could not verify the model, even with the loop invariant.

Temperature control [3]. Our implementation verified this problem after some preliminary transformations. First, we verified that location `shutdown` of the HA is never reached. In order to get a loop invariant, we strengthened the safety property by appending the negation of the guard condition of the transition edge to `shutdown`. The failure of Induction led to a lemma of the form $Q_1 \vee Q_2 \vee Q_3$, but setting each sub-lemma as a loop invariant did not make the verification successful. After some trials, we found that the lemma $Q_1 \wedge (Q_2 \vee Q_3)$ was a necessary loop invariant. Finally, the verification succeeded for $m = 1$ and $n = 1$. This model was also verified efficiently by the MC tools. KeYmaera could not verify the model, even with the loop invariant we had found.

Bouncing ball. This simple nonlinear HA describes a ball with a constant acceleration. As exemplified in [16], we verified that the height of the ball never exceeds the initial energy level of the ball, assuming that the reflection coefficient is smaller than 1. We first attempted the verification under a simple constraint that specified only the sign of each parameter and generated a lemma equivalent to the energy consumption constraint in [16]. We succeeded by setting this lemma as the initial condition and the loop invariant. KeYmaera verified the model given the energy consumption constraint as the initial condition.

European train control system (ETCS) [16,10,2]. The simple model borrowed from [16] is about a train at a position z that should not exceed a limit m. The original model does not have guard conditions so we set them manually based on the analysis in [2]. We attempted to verify the safety property $\Box z < m$ by running the algorithm with $m = 0$ and $n = 1$. Verification succeeded after we obtained a loop invariant from the failure in the validation of VC_1. This model was also verified in [16,10] by using several strategies for the model transformation and loop invariant generation. MC tools could not verify the model because of the nonlinear constraints. KeYmaera verified the model by setting a specific parameter constraint as described in [16].

Highway [14]. This model concerns an autonomous highway with n vehicles. We solved instances for $n = 9$ and $n = 19$, which were also computed by the specific method in [14]. PHAVer verified the instance of $n = 9$ but the computation for $n = 19$ failed after consuming the available memory. KeYmaera could not verify this example.

6.2 Discussions

The MC tools verified three examples quite efficiently. However, our method was better for the other examples. First, it can handle uncertain parameters. Example 1 involves such parameters, as described in Equation (1). In the bouncing ball example, the initial height, velocity, and reflection coefficient are parameterized. Although HyTech and PHAVer verify the same problems with constant values given to the parameters, they cannot verify the instances that involve uncertain parameters. Second, our method scales better: for the highway example, PHAVer can handle only the instances up to $n = 15$ [14].

Although KeYmaera handles various hybrid programs automatically, it did not succeed on most hybrid programs that were translated from hybrid automata. Users often need to annotate models with a loop invariant that might be difficult to extract from the original problem [17]. Otherwise, users need to interact with the underlying theorem prover to investigate the correct derivation tree with various deduction rules. Our approach is limited in verification strategies, i.e., induction and loop unrolling, but the results show that the approach is effective for various examples in practice.

Although our method requires that the executions are lasso shaped (from the point of view of the loop invariant P^+), many examples in the literature can be handled. It, however, requires other verification strategies for the case of compositional and distributed hybrid automata.

7 Related Work

Various tools for the logical analysis of hybrid systems have been proposed. These methods translate hybrid systems into an underlying verification framework, such as STeP [15], PVS [1], SAL [9], Fluctuat [5], and Event-B [2,21]. However, neither the translation nor the verification is fully automated, because some invariants must be added manually, and the theorem provers require some interactions.

Another tool, KeYmaera [18,16], developed by Platzer et al., has been successful in recent years. This tool supports *hybrid programs* that are annotated using *differential (algebraic) dynamic logic*. A dedicated theorem prover verifies the programs by using a set of proof strategies [16]. With its imperative language, which is more expressive than HA, and its corresponding logic (which depends on 141 inference rules [18]), KeYmaera is able to perform various logical analysis through a variety of strategies, including induction, and can serve as a basis for a complete verification framework [16]. In contrast, our framework consists of a light imperative language that is sufficiently expressive to encode HA executions and a logical framework that is introduced to pursue automated verification with the induction strategy.

Recently, a logical analysis tool based on the framework of Hoare logic and relying on infinitesimal variables was proposed [10]. Although its verification scheme comes with several strategies and an invariant generation technique, its practical uses are still unclear.

There are techniques for hybrid systems that generate polynomial invariants by analyzing the executions of a HA via Gröbner basis manipulations [20,19]. These methods could be integrated in the Learn procedure of our framework.

The proposed method also relates to BMC methods. BMC of infinite executions based on induction has been proposed (e.g., [6]), but this approach is applied to discrete systems with continuous states. Most of the BMC tools for hybrid systems handle only finite executions. This is not the case for Hybrid SAL Relational Abstracter [22]. This tool is a translator from hybrid systems to discrete systems with a specific abstraction method. Our method directly handles HA without the abstraction.

8 Conclusions

This paper presents a tool for logical analysis of safety properties of HA, which is able to deal with a large class of linear and nonlinear HA, in contrast with the model-checking approach found in major existing tools.

Rather than introducing various derivation rules to automatically verify HA, we are using a simple process inspired from deductive program verification: strongest postcondition calculus. It allows us to compute logical formulas that, once proved, guarantee the safety of the HA. Our experiments show that our method succeeds in a reasonable time on some example HA from literature, including some that were not solvable with existing tools. The verification process amounts to finding loop invariants, as is the case for program verification. This search for sufficient invariants is guided by the responses from the decision procedures assisted by Mathematica.

A limitation of our approach is that the invariant generation process still requires some human interaction. Efficient automated search of invariant generations is the next challenge for us to tackle. Another direction for further research would be to explore the relation between our approach and some methods from model checking, e.g., verification of an over-approximated model [4].

Acknowledgments. The authors are indebted to the anonymous referees for their helpful comments. This work was partially funded by JSPS (KAKENHI 23-3810).

References

1. Ábrahám-Mumm, E., Steffen, M., Hannemann, U.: Verification of hybrid systems: Formalization and proof rules in PVS. In: ICECCS, pp. 48–57 (2001)
2. Abrial, J.-R., Su, W., Zhu, H.: Formalizing hybrid systems with Event-B. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 178–193. Springer, Heidelberg (2012)
3. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138(1), 3–34 (1995)
4. Alur, R., Henzinger, T.A., Lafferriere, G., Pappas, G.J.: Discrete abstractions of hybrid systems. Proc. of the IEEE 88, 971–984 (2000)
5. Bouissou, O., Goubault, E., Putot, S., Tekkal, K., Vedrine, F.: HybridFluctuat: A static analyzer of numerical programs within a continuous environment. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 620–626. Springer, Heidelberg (2009)
6. de Moura, L., Rueß, H., Sorea, M.: Bounded model checking and induction: From refutation to verification. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 14–26. Springer, Heidelberg (2003)
7. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18(8), 453–457 (1975)
8. Frehse, G.: PHAVer: Algorithmic verification of hybrid systems past HyTech. International Journal on Software Tools for Technology Transfer (STTT) 10(3), 263–279 (2008)
9. Ghosh, R., Tiwari, A., Tomlin, C.J.: Automated symbolic reachability analysis; with application to delta-notch signaling automata. In: Maler, O., Pnueli, A. (eds.) HSCC 2003. LNCS, vol. 2623, pp. 233–248. Springer, Heidelberg (2003)

10. Hasuo, I., Suenaga, K.: Exercises in *nonstandard static analysis* of hybrid systems. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 462–478. Springer, Heidelberg (2012)
11. Henzinger, T.A.: The theory of hybrid automata. Verification of Digital and Hybrid Systems (NATO ASI Series F: Computer and Systems Sciences) 170, 265–292 (2000)
12. Henzinger, T.A., Ho, P.H., Wong-Toi, H.: HyTech: A model checker for hybrid systems. STTT 1, 110–122 (1997)
13. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580, 583 (1969)
14. Jha, S.K., Krogh, B.H., Weimer, J.E., Clarke, E.M.: Reachability for linear hybrid automata using iterative relaxation abstraction. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 287–300. Springer, Heidelberg (2007)
15. Manna, Z., Sipma, H.: Deductive verification of hybrid systems using STeP. In: Henzinger, T.A., Sastry, S.S. (eds.) HSCC 1998. LNCS, vol. 1386, pp. 305–318. Springer, Heidelberg (1998)
16. Platzer, A.: Logical Analysis of Hybrid Systems. Springer (2010)
17. Platzer, A.: Guide for KeYmaera hybrid systems verification tool (2012), `http://symbolaris.com/info/KeYmaera-guide.html` (accessed January 1, 2013)
18. Platzer, A., Quesel, J.-D.: KeYmaera: A hybrid theorem prover for hybrid systems (System description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 171–178. Springer, Heidelberg (2008)
19. Rodríguez-Carbonell, E., Tiwari, A.: Generating polynomial invariants for hybrid systems. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 590–605. Springer, Heidelberg (2005)
20. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constructing invariants for hybrid systems. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 539–554. Springer, Heidelberg (2004)
21. Su, W., Abrial, J.-R., Zhu, H.: Complementary methodologies for developing hybrid systems with event-B. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 230–248. Springer, Heidelberg (2012)
22. Tiwari, A.: HybridSAL relational abstracter. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 725–731. Springer, Heidelberg (2012)

Priced Timed Automata and Statistical Model Checking[⋆]

Kim Guldstrand Larsen

Computer Science, Aalborg University, Denmark

Abstract. The notions of priced timed automata (PTA) and energy games (EG) provide useful modeling formalisms for energy-aware and energy-harvesting embedded systems. We review these formalisms and a range of associated decision problems covering cost-optimal reachability, model-checking and cost-bounded infinite strategies. Decidability of several of these problems require tight bounds on the number of clocks and cost variables. Thus, we turn to statistical model checking (SMC), which has emerged as a highly scalable simulation-based "approximate" validation technique. In a series of recent work we have developed a natural stochastic semantics for PTAs allowing for statistical model checking to be performed. The resulting techniques have been implemented in Uppaal-smc, and applied to the performance analysis of a number of systems ranging from real-time scheduling, mixed criticality systems, sensor networks, energy aware systems and systems biology.

1 Introduction

The model of timed automata, introduced by Alur and Dill [2,3], has by now established itself as a classical formalism for describing the behaviour of real-time systems. A number of important properties has been shown decidable, including reachability, model checking and several behavioural equivalences and preorders.

By now, real-time model checking tools such as Uppaal [9,50] and Kronos [30] are based on the timed automata formalism and on the substantial body of research on this model that has been targeted towards transforming the early results into practically efficient algorithms — *e.g.* [8,14,7,12] — and data structures — *e.g.*[49,48,13,13].

More recently, model-checking tools in general and Uppaal in particular have been applied to solve realistic scheduling problems by a reformulation as reachability problems — *e.g.* [42,43,1,54]. Aiming at *optimal* scheduling, *priced timed automata* [10,6] have emerged as a useful formalism for formulating and solving a broad range of resource allocation problems of importance in applications areas such as, *e.g.*, embedded systems.

⋆ Work partially supported by the VKR Centre of Excellence MT-LAB, the Sino-Danish Basic Research Center IDEA4CPS.

2 Priced Timed Automata

Within the model of priced timed automata, the cost variables serve purely as *evaluation functions* or *observers*, *i.e.*, the behaviour of the underlying timed automatoa may in no way depend on the cost variables. As a consequence of this restriction – and in contrast to the related models of constant slope and linear hybrid automata – a number of optimization problems have been shown decidable for priced timed automata including minimun-cost reachability [10,5,20], optimal (minimum and maximum cost) reachability in multi-priced settings [52].

Dually, computability of cost-optimal infinite schedules have been established covering optimal infinite schedules in terms of minimal (or maximal) *cost per time ratio* in the limit have been obtained in [21,22] and optimal infinite schedules in terms of minimal (or maximal) *discounted total cost* [41].

In terms of tool support Uppaal Cora [46,15,16,57] provides an efficient method for computing cost-optimal or near-optimal solutions to reachability questions, implementing a symbolic A* algorithm based on a new data strucutre (so-called priced zones) allowing for efficient symbolic state-representation with additional cost-information.

Cost-extended versions of temporal logics such as CTL (branching-time) and LTL (linear-time) appear as a natural "generalizations" of the above optimization problems. Just as TCTL and MTL provide extensions of CTL and LTL with time-constrained modalities, WCTL and WMTL are extensions with cost-constrained modalities interpreted with respect to priced timed automata. Unfortunately, the addition of cost now turns out to come with a price: whereas the model-checking problems for timed automata with respect to TCTL and MTL are decidable, it has been shown in [31] that model-checking with respect to WCTL is undecidable for priced timed automata with three clocks or more. In contrast [26,27] shows that model checking with respect to WCTL is decidable under the single clock assumption. Decidability of WCTL for priced timed automata with two clocks is still an open (and hard) problem.

3 Energy Games

In [25] we began the study of a new class of resource scheduling problems, namely that of constructing infinite schedules or strategies subject to boundary constraints on the accumulation of resources, so-called *energy-games* or *energy-schedules*.

More specifically, we consider priced timed automata with *positive* as well as *negative* price-rates. This extension allows for the modelling of systems where resources are not only consumed but also occasionally produced or regained. In [25] three infinite scheduling problems was considered: *lower-bound* where the energy level never must go below zero, *interval-bound* where energy level must be maintained within a given interval, and *weak* upper bound, which does not prevent energy-increasing behaviour from proceeding once the upper bound is reached but merely maintains the energy level at the upper bound.

For one-clock priced timed automata both the lower-bound and the lower-weak-upper-bound problems are shown decidable (in polynomial time) [25], whereas the interval-bound problem is proved to be undecidable in a game setting. Decidability of the interval-bound problem for one-clock priced timed automata as well as decidability of all of the considered scheduling problems for priced timed automata with two or more clocks are still unsettled.

More recently in [24] the decidability of [25] for the lower-bound problem has been extended to the setting of "$1\frac{1}{2}$" priced timed automata and with prices growing either linearly (*i.e.* $\dot{p} = k$) or exponentially (*i.e.* $\dot{p} = kp$) [23]. By "$1\frac{1}{2}$-clock" priced timed automata we refer to one-clock priced timed automata augmented with discontinuous (discrete) updates (*i.e.*, $p := p + c$) of the price on edges: discrete updates can easily be encoded using a second clock but do not provide the full expressive power of two clocks. Surprisingly, the presence of discrete updates makes the lower-bound problem significantly more intricate. In particular, whereas region-stable strategies suffice in the search for infinite lower-bound schedules for one-clock priced timed automata, this is no longer the case when discrete updates are permitted. Not being able to rely on the classical region construction, the key to our decidability result is the notion of an *energy function* providing an abstraction of a path in the priced timed automaton.

In contrast, the existence of interval-constrained infinite runs – where a simple energy-maximizing strategy does not suffice – have recently been proven undecidable for weighted timed automata with varying numbers of clocks and weight variables: e.g. two clocks and two weight variables [56] one clock and two weight variables [40], and two clocks and one weight variable [55]. Also, the interval-constrained problem is undecidable for weighted timed automata with one clock and one weight variable in the game setting [25].

Still, the general problem of existence of infinite lowerbound runs for weighted timed automata has remained unsettled since [25] until the recent paper [28], which close the problem by proving undecidability undecidable for weighted timed automata with four or more clocks. The same paper also considers the variant where only the existence of time-bounded runs are required. In particular it is shown that this restriction makes the problem decidable and NEXPTIME-complete

4 Statistical Model Checking

Statistical Model Checking (SMC) [53,45,58,59,44] is an approach that has recently been proposed as new validation technique for large-scale, complex systems. The core idea of SMC is to conduct some simulations of the system, monitor them, and then use statistical methods (including sequential hypothesis testing or Monte Carlo simulation) in order to decide with some degree of confidence whether the system satisfies the property or not. By nature, SMC is a compromise between testing and classical formal method techniques. Simulation-based methods are known to be far less memory and time intensive than exhaustive ones, and are some times the only option.

In a series of recent works [39,38], we have investigated the problem of Statistical Model Checking for networks of Priced Timed Automata (PTAs), being timed automata, whose clocks can evolve with different rates, while [1] being used with no restrictions in guards and invariants. In [38], we have proposed a natural stochastic semantics for such automata, which allows to perform statistical model checking. Our work has been implemented in Uppaal-smc, providing a new statistical model checking engine for the tool Uppaal. Uppaal-smc relies on a series of extensions of the statistical model checking approach generalized to handle real-time systems and estimate undecidable problems. Uppaal-smc comes together with a rich modeling and specification language [33,32], as well as a friendly user interface that allows a user to specify complex problems in an efficient manner as well as to get feedback in the form of probability distributions and compare probabilities to analyze performance aspects of systems. Also, distributed implementations of the various statistical model checking algorithms has been given with demonstrated linear speed-up [34].

Most recently, we have extended Uppaal-smc to networks of stochastic hybrid automata, allowing clock rates to depend not only on values of discrete variables but also on the value of other clocks, effectively amounting to ordinary differental equations. In particular our original race-based stochastic semantics extends to this setting with the use of Dirac's delta-functions, to allow for the co-existence of (time-wise) stochastic and determinstic components. This extension of Uppaal-smc has already been applied to a wide range of hybrid systems example from real-time scheduling and mixed criticality systems [36], energy aware systems [35] and systems biology [37].

Based on the real-time scheduling problem of [36], we have shown how statistical model checking may serve as an indispensable tool for exhibiting concrete (rare) counter examples witnessing non-schedulability in the setting of stop-watch automata, where the Uppaal verification engine is over-approximate.

The Uppaal-Cora branch [47,17,57] offers an efficient, agent-based and symbolic engine for solving a large range of optimization problems given their model as priced timed automata [11]. However, the tool is restricted to models with a single cost-variable (though extensions have been proposed [51]), with – for decidability – crucial assumption that the cost-variable is only used as an observer (thus cannot be used in guards or invariants). This assumption is lifted slightly in the a sequence of recent work on *energy timed automata* [25,24,29], where the cost-variable is required to be within given bounds. However, in order to achieve decidability strong restrictions on the number of clocks and cost variables are required. We demonstrate how he new SMC engine may provide a competitive and scalable method opening the possibility for optimization to a wider range of models.

[1] In contrast to the usual restriction of priced timed automata [11,4].

References

1. Abdeddaïm, Y., Kerbaa, A., Maler, O.: Task graph scheduling using timed automata. In: IPDPS, p. 237. IEEE Computer Society (2003)
2. Alur, R., Dill, D.L.: Automata for modeling real-time systems. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 322–335. Springer, Heidelberg (1990)
3. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (1994)
4. Alur, R., La Torre, S., Pappas, G.: Optimal paths in weighted timed automata. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC 2001. LNCS, vol. 2034, pp. 49–62. Springer, Heidelberg (2001)
5. Alur, R., La Torre, S., Pappas, G.J.: Optimal paths in weighted timed automata. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC 2001. LNCS, vol. 2034, pp. 49–62. Springer, Heidelberg (2001)
6. Alur, R., Torre, S.L., Pappas, G.J.: Optimal paths in weighted timed automata. In: Benedetto, Sangiovanni-Vincentelli [18], pp. 49–62.
7. Behrmann, G., Bengtsson, J., David, A., Larsen, K.G., Pettersson, P., Yi, W.: Uppaal implementation secrets. In: Damm, W., Olderog, E.-R. (eds.) FTRTFT 2002. LNCS, vol. 2469, pp. 3–22. Springer, Heidelberg (2002)
8. Behrmann, G., Bouyer, P., Larsen, K.G., Pelánek, R.: Lower and upper bounds in zone based abstractions of timed automata. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 312–326. Springer, Heidelberg (2004)
9. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
10. Behrmann, G., Fehnker, A., Hune, T., Larsen, K.G., Pettersson, P., Romijn, J., Vaandrager, F.W.: Minimum-cost reachability for priced timed automata. In: Benedetto, Sangiovanni-Vincentelli [18], pp. 147–161.
11. Behrmann, G., Fehnker, A., Hune, T., Larsen, K.G., Pettersson, P., Romijn, J., Vaandrager, F.: Minimum-cost reachability for priced timed automata. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC 2001. LNCS, vol. 2034, pp. 147–161. Springer, Heidelberg (2001)
12. Behrmann, G., Hune, T., Vaandrager, F.W.: Distributing timed model checking - how the search order matters. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 216–231. Springer, Heidelberg (2000)
13. Behrmann, G., Larsen, K.G., Pearson, J., Weise, C., Yi, W.: Efficient timed reachability analysis using clock difference diagrams. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 341–353. Springer, Heidelberg (1999)
14. Behrmann, G., Larsen, K.G., Pelánek, R.: To store or not to store. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 433–445. Springer, Heidelberg (2003)
15. Behrmann, G., Larsen, K.G., Rasmussen, J.I.: Priced timed automata: Algorithms and applications. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2004. LNCS, vol. 3657, pp. 162–182. Springer, Heidelberg (2005)
16. Behrmann, G., Larsen, K.G., Rasmussen, J.I.: Optimal scheduling using priced timed automata. SIGMETRICS Performance Evaluation Review 32(4), 34–40 (2005)
17. Behrmann, G., Larsen, K.G., Rasmussen, J.I.: Optimal scheduling using priced timed automata. SIGMETRICS Performance Evaluation Review 32(4), 34–40 (2005)

18. Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.): 4th International Workshop on Hybrid Systems: Computation and Control, HSCC 2001. LNCS, vol. 2034. Springer, Heidelberg (2001)
19. Berry, G., Comon, H., Finkel, A. (eds.): 13th International Conference on Computer Aided Verification, CAV 2001. LNCS, vol. 2102. Springer, Heidelberg (2001)
20. Bouyer, P., Brihaye, T., Bruyère, V., Raskin, J.-F.: On the optimal reachability problem on weighted timed automata. Formal Methods in System Design 31(2), 135–175 (2007)
21. Bouyer, P., Brinksma, E., Larsen, K.G.: Staying alive as cheaply as possible. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 203–218. Springer, Heidelberg (2004)
22. Bouyer, P., Brinksma, E., Larsen, K.G.: Optimal infinite scheduling for multi-priced timed automata. Formal Methods in System Design 32(1), 2–23 (2008)
23. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N.: Timed automata with observers under energy constraints (2009) (under submission)
24. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N.: Timed automata with observers under energy constraints. In: Johansson, K.H., Yi, W. (eds.) HSCC, pp. 61–70. ACM (2010)
25. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N., Srba, J.: Infinite runs in weighted timed automata with energy constraints. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 33–47. Springer, Heidelberg (2008)
26. Bouyer, P., Larsen, K.G., Markey, N.: Model-checking one-clock priced timed automata. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 108–122. Springer, Heidelberg (2007)
27. Bouyer, P., Larsen, K.G., Markey, N.: Model checking one-clock priced timed automata. Logical Methods in Computer Science 4(2:9) (June 2008)
28. Bouyer, P., Larsen, K.G., Markey, N.: Lower-bound constrained runs in weighted timed automata. In: QEST, pp. 128–137. IEEE Computer Society (2012)
29. Bouyer, P., Larsen, K.G., Markey, N.: Lower-bound constrained runs in weighted timed automata. In: Proceedings of the 9th International Conference on Quantitative Evaluation of Systems, QEST 2012. IEEE Computer Society Press, London (September 2012) (to appear)
30. Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine, S.: Kronos: A model-checking tool for real-time systems. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 546–550. Springer, Heidelberg (1998)
31. Brihaye, T., Bruyère, V., Raskin, J.-F.: Model-checking for weighted timed automata. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS 2004/FTRTFT 2004. LNCS, vol. 3253, pp. 277–292. Springer, Heidelberg (2004)
32. Bulychev, P., David, A., Larsen, K., Legay, A., Li, G., Poulsen, D.: Rewrite-based statistical model checking of wmtl (under submission)
33. Bulychev, P., David, A., Guldstrand Larsen, K., Legay, A., Li, G., Bøgsted Poulsen, D., Stainer, A.: Monitor-based statistical model checking for weighted metric temporal logic. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18 2012. LNCS, vol. 7180, pp. 168–182. Springer, Heidelberg (2012)
34. Bulychev, P.E., David, A., Larsen, K.G., Mikucionis, M., Legay, A.: Distributed parametric and statistical model checking. In: Barnat, J., Heljanko, K. (eds.) PDMC. EPTCS, vol. 72, pp. 30–42 (2011)
35. David, A., Du, D., Larsen, K.G., Mikučionis, M., Skou, A.: An evaluation framework for energy aware buildings using statistical model checking. Science China, Information Sciences (2012) (submitted)

36. David, A., Larsen, K.G., Legay, A., Mikučionis, M.: Schedulability of herschel-planck revisited using statistical model checking. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012, Part II. LNCS, vol. 7610, pp. 293–307. Springer, Heidelberg (2012)
37. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B., Sedwards, S.: Runtime verification of biological systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012, Part I. LNCS, vol. 7609, pp. 388–404. Springer, Heidelberg (2012)
38. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B., van Vliet, J., Wang, Z.: Statistical model checking for networks of priced timed automata. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 80–96. Springer, Heidelberg (2011)
39. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Wang, Z.: Time for statistical model checking of real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 349–355. Springer, Heidelberg (2011)
40. Fahrenberg, U., Juhl, L., Larsen, K.G., Srba, J.: Energy games in multiweighted automata. In: Cerone, A., Pihlajasaari, P. (eds.) ICTAC 2011. LNCS, vol. 6916, pp. 95–115. Springer, Heidelberg (2011)
41. Fahrenberg, U., Larsen, K.G.: Discount-optimal infinite runs in priced timed automata. Electr. Notes Theor. Comput. Sci. (2008) (to be published)
42. Fehnker, A.: Scheduling a steel plant with timed automata. In: RTCSA, pp. 280–286. IEEE Computer Society (1999)
43. Hune, T., Larsen, K.G., Pettersson, P.: Guided synthesis of control programs using uppaal. Nord. J. Comput. 8(1), 43–64 (2001)
44. Katoen, J.-P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The ins and outs of the probabilistic model checker mrmc. Perform. Eval. 68(2), 90–104 (2011)
45. Laplante, S., Lassaigne, R., Magniez, F., Peyronnet, S., de Rougemont, M.: Probabilistic abstraction for model checking: An approach based on property testing. ACM TCS 8(4) (2007)
46. Larsen, K.G., Behrmann, G., Brinksma, E., Fehnker, A., Hune, T., Pettersson, P., Romijn, J.: As cheap as possible: Efficient cost-optimal reachability for priced timed automata. In: Berry, et al. (eds.) [19], pp. 493–505.
47. Larsen, K.G., Behrmann, G., Brinksma, E., Fehnker, A., Hune, T., Pettersson, P., Romijn, J.: As cheap as possible: Efficient cost-optimal reachability for priced timed automata. In: Berry, et al. (eds.) [19], pp. 493–505.
48. Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: Efficient verification of real-time systems: compact data structure and state-space reduction. In: IEEE Real-Time Systems Symposium, pp. 14–24. IEEE Computer Society (1997)
49. Larsen, K.G., Pearson, J., Weise, C., Yi, W.: Clock difference diagrams. Nord. J. Comput. 6(3), 271–298 (1999)
50. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. STTT 1(1–2), 134–152 (1997)
51. Larsen, K.G., Rasmussen, J.I.: Optimal conditional reachability for multi-priced timed automata. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 234–249. Springer, Heidelberg (2005)
52. Larsen, K.G., Rasmussen, J.I.: Optimal reachability for multi-priced timed automata. Theor. Comput. Sci. 390(2-3), 197–213 (2008)
53. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: An overview. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 122–135. Springer, Heidelberg (2010)
54. Maler, O.: Timed automata as an underlying model for planning and scheduling. In: Fox, M., Coddington, A.M. (eds.) AIPS Workshop on Planning for Temporal Domains, pp. 67–70 (2002)

55. Markey, N.: Verification of Embedded Systems – Algorithms and Complexity. PhD thesis, Ecole Normale Superieure de Chachan (2011)
56. Quaas, K.: On the interval-bound problem for weighted timed automata. In: Dediu, A.-H., Inenaga, S., Martín-Vide, C. (eds.) LATA 2011. LNCS, vol. 6638, pp. 452–464. Springer, Heidelberg (2011)
57. Rasmussen, J.I., Behrmann, G., Larsen, K.G.: Complexity in simplicity: Flexible agent-based state space exploration. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 231–245. Springer, Heidelberg (2007)
58. Sen, K., Viswanathan, M., Agha, G.: Statistical model checking of black-box probabilistic systems. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 202–215. Springer, Heidelberg (2004)
59. Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 223–235. Springer, Heidelberg (2002)

Improved Reachability Analysis in DTMC via Divide and Conquer

Songzheng Song[1], Lin Gui[1], Jun Sun[2], Yang Liu[3], and Jin Song Dong[1]

[1] National University of Singapore
{songsongzheng,lin.gui}@nus.edu.sg, dongjs@comp.nus.edu.sg
[2] Singapore University of Technology and Design
sunjun@sutd.edu.sg
[3] Nanyang Technological University
yangliu@ntu.edu.sg

Abstract. Discrete Time Markov Chains (DTMCs) are widely used to model probabilistic systems in many domains, such as biology, network and communication protocols. There are two main approaches for probability reachability analysis of DTMCs, i.e., solving linear equations or using value iteration. However, both approaches have drawbacks. On one hand, solving linear equations can generate accurate results, but it can be only applied to relatively small models. On the other hand, value iteration is more scalable, but often suffers from slow convergence. Furthermore, it is unclear how to parallelize (i.e., taking advantage of multi-cores or distributed computers) these two approaches. In this work, we propose a divide-and-conquer approach to eliminate loops in DTMC and hereby speed up probabilistic reachability analysis. A DTMC is separated into several partitions according to our proposed cutting criteria. Each partition is then solved by Gauss-Jordan elimination effectively and the state space is reduced afterwards. This divide and conquer algorithm will continue until there is no loop existing in the system. Experiments are conducted to demonstrate that our approach can generate accurate results, avoid the slow convergence problems and handle larger models.

1 Introduction

As an automatic verification technique, model checking [7] has been applied to a variety of domains from hardware to software, and from concurrent systems to probabilistic systems. Different from traditional concurrent systems, probabilistic systems have stochastic characteristics in their behaviors, which means some behaviors follow specific probabilistic distributions. This kind of systems widely exist in many domains, from communication protocols to biology systems. For example, in the randomized leader election protocol [9], multiple processes want to elect one leader. Each process will first randomly choose a natural number from a specific range as its id. The process with a unique highest id will be elected as a leader. If several processes have the same highest id, the selection procedure will repeat. Therefore uniform distribution is necessary in this system. As a result, model checking probabilistic systems is an important topic in formal verification.

Discrete Time Markov Chain (DTMC) is a widely used formalism in probabilistic model checking. The difference between DTMC and traditional Labeled Transition System (LTS) is that non-determinism in LTS is replaced by probabilistic choices in DTMC. In a DTMC, at each step the transition from one state to another must follow specific probability distributions, and for each state there is exactly one probability distribution for the successor states. Reachability analysis plays a key role in DTMC verification, e.g., it is used to decide the probability of reaching certain disastrous state. Verification of properties such as Probabilistic Computational Tree Logic (PCTL) and Linear Temporal Logic (LTL) can be reduced to the reachability analysis problem [5]. E.g., for LTL properties a product construction with a deterministic Rabin/Muller-automaton is needed to obtain the target states. Therefore in this work we focus on improving reachability analysis in DTMC verification.

Given the transition relation of a DTMC, the transition probability matrix from one state to another can be built. After the target states are decided, each state in the matrix can be represented by a variable, which means the probability of reaching the target states from this state. Next, there are mainly two approaches to calculate the probability from initial states to the targets. One is solving linear equations directly. In this method, variables representing intermediate states (which are not target or initial) are eliminated gradually through equations operation, and finally variables representing the initial states' probability of reaching targets can be solved. The other approach is using value iteration method, which works by finding a better approximation iteratively until certain stopping criteria are satisfied. The approach based on solving linear equations is straightforward to understand and it guarantees to deliver accurate result. However, since we need one variable for each state in the system, a lot of variables are needed for large systems whereas state-of-the-art linear solvers are limited to thousands of variables only. Therefore the applicability of this approach is limited to small-scale systems. On the other hand, the value iteration method tries to find fix-points iteratively, and it has relatively better performance in handling systems with a large number of states. Therefore it is more popular in probabilistic model checkers such as PRISM [12] and MRMC [10,11]. However, this approach also has its drawback: slow convergence, i.e., it may take a large number of iterations before the approximations converge to a certain value. The phenomenon exists when there are complicated loops existing in the probabilistic systems, although the state space of such systems may not be very huge. The number of iterations is related to the subdominant eigenvalue of the probability transition matrix [18].

To tackle the above-mentioned problems, in this work we propose a new approach to verify DTMC models, especially for the ones with loops using a divide-and-conquer strategy. Instead of directly calculating the probability from initial states to targets, we divide the whole state space into several partitions, and solve them individually to eliminate loops. Afterwards, the remaining acyclic DTMC can be solved efficiently via value iteration method.

As we mentioned above, the slow convergence problem in value iteration comes from loops. Therefore, the first step of our approach is finding Strongly Connected Components (SCCs). This SCC-based approach is similar to previous work such as [3,6,1,13]. However, instead of using SCC's topology order [6,13], we solve each SCC independently by calculating the new transition probability from input states to output states of the SCC, which is similar to work [3,1]. These new transitions are denoted as *abstract* transitions since SCCs are abstracted by transitions from input states to output states. However, [1] focuses on counterexample generation and abstracts SCCs via iteratively finding the smallest SCCs. On the contrary, we divide each SCC having a large number of states to several smaller partitions. For each partition, abstract transitions from its input to output are calculated via solving linear equations. Here we use Gauss-Jordan elimination [2]. Further, the states in each partition which are not input states will be removed, and thus the states in the SCC can be reduced. Afterwards, the new SCC is ready for next iteration of divide and conquer. This procedure for each SCC will be done iteratively until any of the following three criteria is satisfied. First, there is no more loop in the reduced SCC. Then this part will be left alone since it is already acyclic. Second, the number of remaining states in reduced SCC is small enough to be solved via a linear solver. Third, the last iteration does not reduce any states. In the second and third scenarios, the final SCC will be solved via linear equation again, and final abstract transitions will be generated. After all loops in SCCs are resolved, the whole DTMC becomes acyclic, and value iteration is used to calculate the probability from initial states to targets. Since the abstract transitions from each partition's input states to output states are determined by the partition itself and independent to other partitions, multi-cores or distributed computers can be straightforwardly used here to solve each partition simultaneously, which makes the verification faster.

Contributions Compared with previous work, our contribution is threefold, as we summarize below.

1. A new divide-and-conquer approach for DTMC reachability analysis is proposed, which combines solving linear equations and value iteration methods together and tackles the problem that huge loops make the DTMC verification inefficient.
2. Based on the fact that each SCC and even each group in one SCC is independent from others, we use parallel computation to further speed up the verification.
3. The new approach has been implemented into our model checking framework PAT, and several representative experiments are conducted to show the effectiveness of our approach.

Organization The paper is structured as follows. Section 2 recalls relative background. In Section 3, we introduce our algorithm in details. The evaluation is reported in Section 4. Section 5 surveys related work and concludes the paper.

2 Preliminaries

In this section, we recall some background knowledge, which is relevant in the rest of this paper.

2.1 Discrete Time Markov Chains

Discrete Time Markov Chains (DTMCs) are widely used in modeling stochastic systems. Meanwhile, time requirement in DTMC is discrete. Without loss of generality,

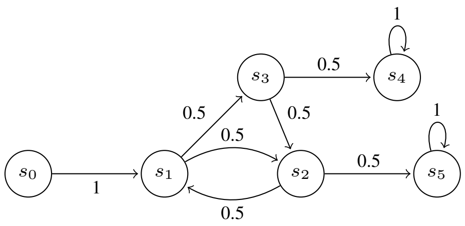


Fig. 1. An Example of SCC

we have the following two assumptions in this work. 1) There is only one initial state in the whole system and 2) DTMC is deadlock free. It is known that a deadlock state in a DTMC can add a self-loop having probability 1 without affecting the calculation result. The formal definition of DTMC is as follows.

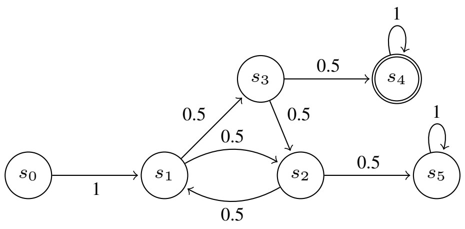
Definition 1. *A Discrete Time Markov Chain is a tuple $\mathcal{M}$ = (S, s_{init}, Tr, AP, L) where S is a set of states; $s_{init} \in S$ is the initial state of the system; $Tr : S \times S \to [0, 1]$ is the probability transition relation between states, which satisfies $\forall s \in S$, $\Sigma_{s' \in S} Tr(s, s') = 1$; AP is a set of atomic propositions and L: $S \to 2^{AP}$ is a labeling function.*

An infinite or a finite *path* in $\mathcal{M}$ is defined as a sequence of states $\pi = \langle s_0, s_1, \cdots \rangle$ or $\pi = \langle s_0, s_1, \cdots, s_n \rangle$ respectively, such that $\forall i \leq 0$ (for finite paths, $i \in [0, n-1]$), $Tr(s_i, s_{i+1}) > 0$. The probability of exhibiting π in $\mathcal{M}$ is $\mathcal{P}_{\mathcal{M}}(\pi) = Tr(s_0, s_1) \times Tr(s_1, s_2) \times Tr(s_2, s_3) \times \cdots$. Given a set of paths Π of $\mathcal{M}$, $\mathcal{P}_{\mathcal{M}}(\Pi) = \sum_{\pi \in \Pi} \mathcal{P}_{\mathcal{M}}(\pi)$.

A set of states $C \subseteq S$ is called *connected* in $\mathcal{M}$ iff $\forall s, s' \in C$, there is a finite path $\pi = \langle s_0, s_1, \cdots, s_n \rangle$ satisfying $s_0 = s \wedge s_n = s' \wedge \forall i \in [0, n], s_i \in C$. Strongly Connecte Components (SCCs) are those maximal sets of states which are mutually connected. An SCC is called *trivial* if it just has one state without a self-loop. An SCC is *nontrivial* iff it is not trivial. A DTMC is *acyclic* iff it only has *trivial* SCCs. Note that one state can only be in one SCC. In other words, SCCs are disjoint. In addition, we define an *adjacent group* (AG) $D \subseteq S$ such that $\exists s \in D, \forall s' \in D \wedge s' \neq s$, there is a finite path $\pi = \langle s_0, s_1, \cdots, s_n \rangle$ satisfying $s_0 = s \wedge s_n = s' \wedge \forall i \in [0, n], s_i \in D$, and s is called *root* state in D. In the following, we refer to adjacent groups simply as groups. The difference between these conceptions is illustrated by the example in Figure 1.

In Figure 1, $\{s_1, s_2\}, \{s_1, s_2, s_3\}$ are *connected*; $\{s_0\}$, $\{s_4\}$, $\{s_5\}$ and $\{s_1, s_2, s_3\}$ are the *SCCs* in the model; AGs are more complex, for example, $\{s_0, s_1, s_2\}$ and $\{s_1, s_2, s_5\}$ are AGs and there are other possible combinations. Note that a set of states like $\{s_0, s_1, s_4\}$ is not a valid AG because there is no root state. *Connected* subgraphs are AGs but the reverse is not always true, e.g., $\{s_0, s_1, s_2\}$ is an AG but not a *connected* subgraph.

Similar to [3,1], in a DTMC $\mathcal{M}$ = (S, s_{init}, Tr, AP, L), given a group of states $\mathcal{D} \subseteq S$, the input states of $\mathcal{D}$ are defined as the states in $\mathcal{D}$ having incoming transitions from states outside $\mathcal{D}$; the output states of $\mathcal{D}$ are defined as states outside $\mathcal{D}$ which have incoming transitions from states in $\mathcal{D}$. Formal definitions are as follows.



$$\begin{cases} p_0 = p_1 \\ p_1 = 0.5 \times p_2 + 0.5 \times p_3 \\ p_2 = 0.5 \times p_1 + 0.5 \times p_5 \\ p_3 = 0.5 \times p_2 + 0.5 \times p_4 \\ p_4 = 1 \\ p_5 = 0 \end{cases}$$

Fig. 2. Reachability Analysis

$$Inp(\mathcal{D}) = \{s' \in \mathcal{D} \mid \exists s \in S \backslash \mathcal{D}.Tr(s, s') > 0\}^1$$
$$Out(\mathcal{D}) = \{s' \in S \backslash \mathcal{D} \mid \exists s \in \mathcal{D}.Tr(s, s') > 0\}$$

2.2 Reachability Analysis

One critical question for quantitative analysis of DMTC models is to compute the probability of reaching a certain set of target states G from the initial state. Here $\Diamond G$ is used to denote the event of reaching G, and $\mathcal{P}_{\mathcal{M}}(s_{init} \models \Diamond G)$ represents the probability that G can be reached from initial state in a DTMC $\mathcal{M}$. Here $\mathcal{P}_{\mathcal{M}}$ can be written as $\mathcal{P}$ if $\mathcal{M}$ is clear. Let $\pi = \langle s_0, s_1, \cdots, s_n \rangle$ represent any finite path in $\mathcal{M}$. Then we have

$$\mathcal{P}(s_{init} \models \Diamond G) = \mathcal{P}(\{\pi \mid s_0 = s_{init} \wedge \exists i \in [0..n], s_i \in G \wedge \forall j \in [0..i-1], s_j \notin G\})$$

Given the transition relation Tr of $\mathcal{M}$, there are two approaches to calculate $\mathcal{P}(s_{init} \models \Diamond G)$. One is solving linear equations, while the other is using value iteration. We use p_i to represent the probability from state s_i to the targets. In the following we use the example in Figure 2 to show how these two approaches work. Note that state s_4 is the only target state, denoted by double cycles.

Solving Linear Equations From the model, the transition matrix between states can be built. For example, $p_1 = 0.5 \times p_2 + 0.5 \times p_3$ and $p_0 = p_1$. Since s_4 is target, p_4 = 1. s_5 cannot reach target obviously, therefore $p_5 = 0$. From these equations, each p_i can be solved through matrix operations. Although this approach can get accurate result, it has drawbacks. Because each state is represented by a variable, there may be a huge number of variables in large scale systems. The state-of-the-art linear solvers are limited to handle thousands of variables, therefore linear equation approach may not be scalable.

Using Value Iterations In this approach, p_i is calculated iteratively. Assume p_i^k is an approximation of p_i after the k-th iteration. Starting from the target state s_4, in k-th iteration we update the probability of states which could reach s_4 in exactly k steps. Obviously, $\forall i \in [0, 3], p_i^0 = 0$. As p_4^k = 1 and p_5^k = 0 for any k, k is ignored in these two states. In the first iteration, p_3 can be updated, and p_3^1 = $\{0.5 \times p_2^0 + 0.5 \times p_4\} = 0.5$; in

[1] If $s_{init} \in \mathcal{D}$, then $s_{init} \in Inp(\mathcal{D})$.

the second iteration, p_1 is updated since s_1 reaches s_3 in one step. It is trivial to show $p_1^2 = \{0.5 \times p_3^1 + 0.5 \times p_2^1\} = 0.25$. In the third iteration, both p_0 and p_2 can be updated since they can reach s_1 in one step. Afterwards, p_3 is updated again because of the update of p_2. Iteratively, p_i in the long run can be calculated. A user-defined threshold is usually necessary to terminate the calculation, according to the desired precision. The result of p_i will be approximated gradually. This approach has better scalability than the linear equations method, so it is more popular in existing model checkers. However, the existence of loops may make the convergence slow. The probability of each state in SCCs will be updated many times, which means a large number of iterations may be needed before the results satisfy the terminating criteria.

2.3 States Abstraction and Gauss-Jordan Elimination

Here we follow the idea of [1]. Given a DTMC $\mathcal{M} = (S, s_{init}, Tr, AP, L)$ and a group of states $\mathcal{D} \subseteq S$, $\mathcal{D}$ can be abstracted by calculating the transition probability from $Inp(\mathcal{D})$ to $Out(\mathcal{D})$. According to the proof in [1], the abstraction of any arbitrary set of states is independent from others, and the abstract transitions **do not affect** the probability of reaching target states G.

One example of the abstraction is in Figure 3. Figure 3 (a) is the original DTMC, which has one SCC $\mathcal{D} = \{s_1, s_2, s_3\}$. $Inp(\mathcal{D}) = \{s_1\}$ and $Out(\mathcal{D}) = \{s_4, s_5\}$. In order to abstract $\mathcal{D}$[2], the probability from $Inp(\mathcal{D})$ to each state $s_{out} \in Out(\mathcal{D})$ should be calculated. Theoretically, the calculation from an SCC's inputs to outputs can be solved via linear equations or value iteration approaches[3]. However, for value iteration approach, since there could be several output states in $Out(\mathcal{D})$, we have to separately calculate the probability from input states to each output state. If there are many output states, this method could be inefficient. In addition, the existence of loops still causes slow convergence issue. Furthermore, using value iteration, there will be some errors because of the user-defined precision, but there is no way to know the error bounds. Therefore, we use a specific linear equation solving technique: Gauss-Jordan elimination [2] to do the abstraction.

Gauss-Jordan elimination is an algorithm for getting matrices in reduced row echelon form that placing zeros above and below each pivot [2]. Here, we briefly introduce how it works in our setting.

Assume there are m states in a set of states, say $\mathcal{D}$, and $|Out(\mathcal{D})| = n$. Then two matrices A and B, containing linear equations information of all transitions in $\mathcal{D}$, are first introduced as follows.

$$A(i,j) = \begin{cases} 1, & \text{if } i = j; \\ -Tr(i,j), & \text{otherwise.} \end{cases} \qquad B(i,k) = -Tr(i,k).$$

Here, A is an $m \times m$ square matrix. $A(i,j)$ is a negative value of probability of transition from i^{th} state to j^{th} state in $\mathcal{D}$ if $i \neq j$. The diagonal elements of A are filled by 1.

[2] Here we take an SCC as an example. Actually this abstraction can be applied to arbitrary set of states, according to [1].

[3] Different from our previous discussion which focuses the calculation from the initial state to targets, here we discuss the probability from input states to every output state of an SCC.

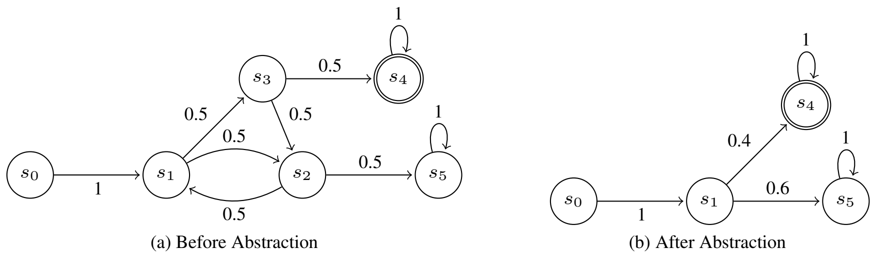


Fig. 3. States Abstraction via Gauss-Jordan Elimination

This records the transition relationship within $\mathcal{D}$. B is an $m \times n$ matrix to record the transition relationship from $\mathcal{D}$ to $Out(\mathcal{D})$. k represents the k^{th} state in $Out(\mathcal{D})$.

Next, augmenting the square matrix A with matrix B, we will have $[A \mid B]$. Gauss-Jordan elimination on $[A \mid B]$ will then produces $[I \mid C]$. Here, I is the identity matrix with 1s on the main diagonal and 0s elsewhere. The new transition probability e.g., $Tr'(i,k)$, stores the transition probability from i^{th} state in $\mathcal{D}$ and k^{th} state in $Out(\mathcal{D})$, which is actually $-C(i,k)$. Now take Figure 3 (a) as an example. Its $[A \mid B]$ and resulting $[I \mid C]$ are listed as follows. In this example, $A(i,j)$ corresponds to $Tr(s_{i+1}, s_{j+1})$ and $B(i,k)$ indicates $Tr(s_{i+1}, s_{k+4})$.

$$[A|B] = \left[\begin{array}{ccc|cc} 1 & -0.5 & -0,5 & 0 & 0 \\ 0 & 1 & -0.5 & 0 & -0.5 \\ 0 & -0.5 & 1 & -0.5 & 0 \end{array}\right] ; \ [I|C] = \left[\begin{array}{ccc|cc} 1 & 0 & 0 & -0.4 & -0.6 \\ 0 & 1 & 0 & -0.2 & -0.8 \\ 0 & 0 & 1 & -0.6 & -0.4 \end{array}\right]$$

Here the transitions from all the states in $\mathcal{D}$ to $Out(\mathcal{D})$ are obtained. Note that those states which are not in $Inp(\mathcal{D})$ will be removed. Therefore we are just interested in the new transitions from $Inp(\mathcal{D})$ to $Out(\mathcal{D})$, which are

$$Tr'(s_1, s_4) = 0.4; \quad Tr'(s_1, s_5) = 0.6;$$

We can obtain that $p_1 = 0.4 \times p_4 + 0.6 \times p_5$ in the abstracted DTMC, which is shown in Figure 3 (b). Given a group of states $\mathcal{D}$, this abstraction procedure is defined as a method $Abs(\mathcal{D})$.

Note that in practice, most transition matrices in probabilistic model checking have a very sparse structure that contains a large number of zeros. We adopt a compressed-row representation [14] as a data structure for matrices in Gauss-Jordan elimination.

3 Divide and Conquer Approach

From the analysis in Section 2, for a large DTMC with complicated loop structure, both linear equations and value iteration method are ineffective, even unworkable. In this section, we propose a divide and conquer approach which tackles the above-mentioned problem. Our main idea is similar to work [3,1], which transfers the original DTMC to an acyclic one by abstracting SCCs recursively so as to reduce the number of state and loops.

Algorithm 1. Divide and Conquer Approach

```
input  : A DTMC $\mathcal{M}$ = $(S, s_{init}, Tr, AP, L)$, target states $G \subseteq S$ and a Bound $B$
output: $\mathcal{P}(s_{init} \models \Diamond G)$
1  Let $\mathcal{C}$ be the set of all nontrivial SCCs in $\mathcal{M}$;
2  while $|\mathcal{C}| > 0$ do
3      Let $\mathcal{D} \in \mathcal{C}$;
4      if $|\mathcal{D} \leq B| \vee Out(\mathcal{D}) \leq 1$ then
5          $Abs(\mathcal{D})$ and $\mathcal{C} \leftarrow \mathcal{C} \backslash \mathcal{D}$
6      else
7          Divide $\mathcal{D}$ into a set of AGs denoted as $\mathcal{A}$;
8          for each $\mathcal{E} \in \mathcal{A}$ do $Abs(\mathcal{E})$;
9          Let $\mathcal{D}'$ be the set of remaining states in $\mathcal{D}$;
10         if $|\mathcal{D}'| \leq B \vee |\mathcal{D}'| = |\mathcal{D}|$ then
11             $Abs(\mathcal{D}')$ and $\mathcal{C} \leftarrow \mathcal{C} \backslash \mathcal{D}$
12         else
13             Let $\mathcal{C}_{\mathcal{D}'}$ be the set of all nontrivial SCCs in $\mathcal{D}'$;
14             $\mathcal{C} \leftarrow (\mathcal{C} \backslash \mathcal{D}) \cup \mathcal{C}_{\mathcal{D}'}$;
15 return $VI(\mathcal{M}, G)$;
```

Intuitively, our approach divides large SCCs into smaller partitions, each of which will be solved via Gauss-Jordan elimination independently. Through this approach, loops will be eliminated. Afterwards, value iteration method is used to decide the final probability of reaching targets. In the following, we introduce our algorithm in details.

3.1 Overall Algorithm

Given a DTMC $\mathcal{M}(S, s_{init}, Tr, AP, L)$ and target states $G \subseteq S$, the probability of reaching G, denoted as $\mathcal{P}(s_{init} \models \Diamond G)$, can be solved by Algorithm 1. Note that B is an input parameter, which indicates SCCs having more than B states should be divided. $Abs(K)$ is defined in Section 2.3. $VI(\mathcal{M}, G)$ indicates calculating the probability of reaching G via value iteration. The procedure of the algorithm is explained in the following.

- The first step is to find all SCCs $\mathcal{C}$ in $\mathcal{M}$ by Tarjan's approach [17], and their input and output states are recorded as well. This is captured by Line 1.
- For each SCC $\mathcal{D} \in \mathcal{C}$, we will first check whether $|\mathcal{D}|$ exceeds B or whether $|Out(\mathcal{D})| > 1$. If not, $Abs(\mathcal{D})$ will be executed directly. States in $\mathcal{D}$ but not in $Inp(\mathcal{D})$ will be removed. Afterwards $\mathcal{D}$ will be removed from $\mathcal{C}$, as shown in Lines 4-5. The reason why we directly abstract cases $|Out(\mathcal{D})| \leq 1$ is as follows.
  - If $|Out(\mathcal{D})| = 0$, $\mathcal{D}$ has no outgoing transitions, then no matter whether $\mathcal{D}$ has target states or not, we do not need to solve $\mathcal{D}$. If $\mathcal{D} \cap G = \phi$, it is obvious that all states in $\mathcal{D}$ has probability 0 to reach G; otherwise, it is trivial to show that all states in $\mathcal{D}$ has probability 1 to reach G.

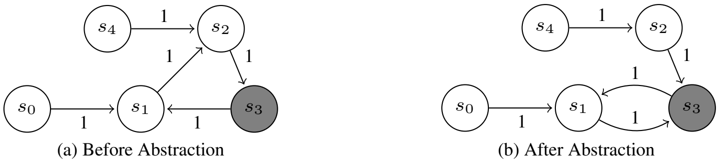


Fig. 4. Destruction of SCC during Abstraction

- If $|Out(\mathcal{D})| = 1$, assume s_{out} is the output state. All paths entering $\mathcal{D}$ will leave it eventually. Therefore, for every $s_i \in Inp(\mathcal{D})$, the probability of paths entering $\mathcal{D}$ via s_i, staying in $\mathcal{D}$ and exiting $\mathcal{D}$ to s_{out} should be 1. So $\mathcal{D}$ can be abstracted directly.

– Lines 7-14 describe the case when $\mathcal{D}$ needs to be divided, i.e., when the SCC has more than B states. First we divide $\mathcal{D}$ into several groups based on some heuristics, each of which has a reasonably small number of state, i.e., less than B. Therefore, for each group $\mathcal{E}$ we use $Abs(\mathcal{E})$ to get the abstraction. Here we choose AG as the structure of each partition, because the existence of the root state, say s_r, may remove the most states after abstraction. In the extreme case where $Inp(\mathcal{E}) = \{s_r\}$, all states in $\mathcal{E}$ except s_r can be removed.

– By removing the states which are not input states of any $\mathcal{E}$, the number of states in $\mathcal{D}$ is often (not always) reduced. Line 10 checks two situations. 1) the size of $\mathcal{D}'$ is smaller than or equal to B, and 2) there is no reduction for $\mathcal{D}$ in this iteration. If 1) is true, then there is no need to divide $\mathcal{D}'$ again, and $Abs(\mathcal{D}')$ is executed directly. If 2) is true, i.e., no state is reduced after divide and conquer, the main reason should be that each state in $\mathcal{D}$ has a lot of pre-states. Therefore every state in one group is an input state and cannot be removed. In this case, $\mathcal{D}'$ should also be abstracted. Afterwards, $\mathcal{D}$ is removed from $\mathcal{C}$. If 1) and 2) are both false, Lines 13-14 will be executed.

– Because of the abstraction, $\mathcal{D}$ may not be an SCC now. An example is shown in Figure 4. On the left hand side, $\mathcal{D} = \{s_1, s_2, s_3\}$; if we group s_1 and s_2 together, then s_3 is this group's output. It is easy to get the abstract transitions between them, as shown in right hand side. Because both s_1 and s_2 are input states, no state is removed. However, it is obvious that $\mathcal{D}' = \{s_1, s_2, s_3\}$ is not an SCC anymore. Tarjan's algorithm is used again to find new SCCs in the $\mathcal{D}'$, captured by Line 13. New SCCs will be added to $\mathcal{C}$ for another iteration.

– When the iteration terminates, there is only trivial SCCs in $\mathcal{M}$ now; in other words, $\mathcal{M}$ is acyclic. Value iteration approach can be used to calculate the probability from the initial state to targets efficiently, and this is captured by Line 15.

As we mentioned in Section 2.3, the iterative abstraction will not affect the final result of the probability calculation. The following theorem establishes that the algorithm is always terminating.

Theorem 1. *Given a finite state DTMC $\mathcal{M}$, Algorithm 1 always terminates.*

Proof. We assume $\hat{S} = \Sigma_{\mathcal{D} \in \mathcal{C}} |\mathcal{D}|$, in other words, $\hat{S}$ is the total number of states in $\mathcal{C}$. Then the theorem can be proved by showing (1) $\hat{S}$ is finite at the beginning, and (2) $\hat{S}$ monotonically decreases after each iteration.

(1) is obviously true because $\mathcal{M}$ has finite number of states, and $\hat{S} \leq |S|$ where S is the set of states of $\mathcal{M}$.

Given an SCC $\mathcal{D} \in \mathcal{C}$, if it satisfies the condition in Line 4, then $\mathcal{D}$ will be removed from $\mathcal{C}$, thus $\hat{S}$ is reduced. Otherwise, from Line 6, there are two possible outputs. (i) $\exists \mathcal{E} \in \mathcal{A}$, $Abs(\mathcal{E})$ reduces its number of states, or (ii) $\forall \mathcal{E} \in \mathcal{A}$, $Abs(\mathcal{E})$ does not reduce its number of states. If (i) is true, then $\hat{S}$ is also reduced. If (ii) is true, then $|\mathcal{D}'| = |\mathcal{D}|$. According to Line 8, $\mathcal{D}$ will be abstracted directly and be removed from $\mathcal{C}$. Thus $\hat{S}$ is still reduced. Therefore (2) is true, and the theorem holds. □

3.2 Dividing Strategies

Although the divide-and-conquer approach is correct and terminating, its efficiency is highly dependent on how an SCC is divided. Assume $\mathcal{A}$ is the set of partitions after dividing an SCC, then a suitable partition, say $\mathcal{E} \in \mathcal{A}$, should satisfy the following conditions.

1. $\mathcal{E}$ should not have too many states, since each partition is abstracted using Gauss-Jordan elimination which is limited to a relatively small number of states;
2. $\mathcal{E}$ should not have too few states as well, otherwise there will be too many partitions to be solved, and the states reduction for $\mathcal{E}$ is inefficient;
3. The smaller $|Out(\mathcal{E})|$ is, the better reduction is achieved. Too many output states will make the input states of $\mathcal{E}$ have too many abstract transitions, which makes the remaining structure complicated, and affects the efficiency of the following abstraction.

As a result, the remaining issue is that given an SCC $\mathcal{D}$, is there any *optimal* strategy to divide it into *suitable* AGs? In practice, the structure of $\mathcal{D}$ could be arbitrary. This increases the difficulty of finding a general strategy for all cases.

The simplest division method is to try to set each AG to have the same number of states. Assume each AG should have N states. Then starting from one input state of $\mathcal{D}$, depth first search (DFS) or breadth first search (BFS) can be used to group every N states together. Afterwards, each AG can be abstracted, and the remaining states are combined together to do the next iteration. The advantage of this strategy is that the number of states in each partition is easily controlled. It can be very efficient in cases where the states in $\mathcal{D}$ has few transitions. However, this method cannot control the number of output states of each partition, and a predefined N may not be suitable for $\mathcal{D}$'s structure.

Therefore, another improved strategy is used to automatically decide the number of states in each AG. Instead of picking a constant N in the beginning, we set a lower bound B_L and an upper bound B_U for each partition. Thus the number of states in each partition should be between B_L and B_U. At first, B_L states will be grouped into $\mathcal{E}$, and $|Out(\mathcal{E})|$ is recorded. Afterwards, some states in $Out(\mathcal{E})$ are added into $\mathcal{E}$, and $|Out(\mathcal{E})|$ is updated. If $|Out(\mathcal{E})|$ keeps unchanged or even becomes smaller after the update, we will try to add more states into $\mathcal{E}$ again. If $|Out(\mathcal{E})|$ is increased but the increase is not significant, a few states will be added into $\mathcal{E}$ but the number should be small. Otherwise $\mathcal{E}$ is confirmed and ready for $Abs(\mathcal{E})$. Note the number of states in $\mathcal{E}$ should be always below B_U. This strategy guarantees

1. the number of states in $\mathcal{E}$ is under control. B_L and B_U guarantee that the size of $\mathcal{E}$ should not be too large or too small.
2. the outputs of $\mathcal{E}$ are also manageable. This guarantees the states structure after abstraction is not too complicated, and is suitable for next iteration.

Parameters B, N, B_L and B_U can be adjusted according to the specific DTMC to get the optimal efficiency.

3.3 Parallel Computation

Previous work such as [6,13] depends on the topological order between different SCCs. Therefore, parallel computation is not so easy to use in their setting. On the contrary, our algorithm eliminates loops via abstracting every SCC one by one, without considering their order. The independence between different SCCs can be proved following the proof in [1]. What is more, even each AG in one SCC is also independent from others, and the proof actually follows the same idea of SCC's independence. Thus, parallelization is suitable in our setting in order to solve different AGs simultaneously.

In details, after finding all SCCs, they are stored with their input and output states. For each SCC, a spare thread can be used to solve it. Therefore, Lines 2-14 in Algorithm 1 can be solved via parallel computation. In addition, whenever an AG is grouped, another spare thread, if there is any, can be used to abstract it. Thus Line 8 in Algorithm 1 can also be handled in parallel.

4 Implementation and Evaluation

We have implemented the algorithm into our model checking framework PAT [15], which supports explicit probabilistic model checking [16] and can be freely downloaded at `http://www.patroot.com`.

In the following, several experiments are conducted to show the efficiency of our new approach. Note that we show the improvement via comparing to PAT itself, which was based on value iteration method previously. Since the only difference between these two versions is the algorithm of reachability analysis, it is fair to check the effectiveness of the new method. Besides, several cases used in our experiment have dynamically updated probabilistic distributions, and the modeling of them by other model checkers is highly nontrivial.

In these experiments, we use the **improved** dividing strategy, and B, B_L, B_U are set to be 300, 100, 150 respectively. In other words, an SCC with more than 300 states should be divided; each group has states between 100 and 150. These parameters are manually selected based on our experimental experience, i.e., generally these parameters have better performance compared with others. The testbed is a server running Windows Server 2008 64 Bit with Intel Xeon 4-Core CPU$\times$2 and 32 GB memory.

First, we use a simple example to show that our approach gets accurate results, resolves the slow convergence problem and results in huge speedup. Assume there are

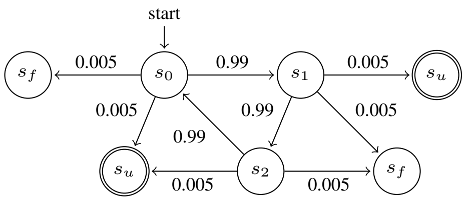


Fig. 5. A Simple Example: N = 3. s_u and s_f are copied for better demonstration.

Table 1. Experiments: A Simple Example

System	PAT (w)			PAT (w/o)		
	Prob	Time (s)	Memory (MB)	Prob	Time (s)	Memory (MB)
N = 500	0.5	0.03	71	0.49987	0.5	24
N = 5000	0.5	0.3	83	0.49987	5.5	63
N = 50000	0.5	2.6	151	0.49987	125.2	111
N = 500000	0.5	29.7	885	0.49987	1612.8	838

$N + 2$ states $\{s_0, s_1, ..., s_{N-1}, s_u, s_f\}$ existing in this example. Each state $s_i, i \in [0..N-1]$, has probability 0.99 to reach $s_{(i+1)\%n}$, and also has probability 0.005 to reach s_u and s_f separately. The case $N = 3$ is shown in Figure 5. Obviously, all states $s_i, i \in [0..N-1]$ compose an SCC, and s_u and s_f are this SCC's outputs. We check the probability from s_0 to s_u, and several experiments are executed based on different value of N as listed in Table 1.

In Table 1, columns $Prob$ represents the probability returned by the model checking algorithms. Columns $PAT(w)$ ($PAT(w/o)$) show the experimental information taken with (without) the new approach. Columns $Time$ represent the total time cost in the verification. For these cases, our new approach outperforms value iteration approach dramatically by reducing the verification time to less than 10%. On the other hand, the memory used in new approach is higher than that used in the previous method, which is reasonable since solving linear equations consumes more memory than value iteration approach. Through the manual analysis, we know that 0.5 is the accurate result while 0.4998 is only an approximation.

Next, we apply our approach to several more meaningful systems and demonstrate that our approach can still improve the efficiency significantly.

In multi-agent systems, dispersion games [8] represent an important scenario, i.e., dispersion games are the generalization of anti-coordination games to an arbitrary number of players and actions. Here we use two strategies designed for dispersion games: *bisic simple strategy* (BSS) and *extend simple strategy* (ESS). BSS assumes the number of players and the number of actions are the same, while ESS does not have this assumption. In each round of the game, every player chooses one action following specific probabilistic distribution, which is updated roundly according to the output of last round. There is a desired outcome in this game called Maximal Dispersion Outcome (MDO), and one property is to calculate the probability that MDO can be achieved.

Another case used in our experiments is coin flipping protocol for polynomial randomized consensus [4] (CS). This case focuses on modeling and verifying the shared

Table 2. Experiments: Benchmark Systems

System	States	Prob	PAT (w)			PAT (w/o)		
			Time (s)	BMR	Memory (MB)	Time (s)	BMR	Memory (MB)
BSS (4)	4196	1	1.3	92.3%	39	0.2	50%	35
BSS (5)	49572	1	3.5	94.3%	297	4.4	11.4%	142
BSS (6)	605890	1	41.4	72.7%	1297	105.3	6.7%	417
BSS (7)	7462639	1	1671	30.1%	6350	2073.1	4.1%	5039
ESS (6, 4)	32662	1	1.4	92.8%	16.3	2.7	14.8%	5.6
ESS (6, 5)	162945	1	6.7	91.1%	48.5	11.4	16.7%	13.9
ESS (7, 5)	463460	1	27.9	84.9%	310	75.8	7.1%	292
ESS (8, 5)	1114480	1	70.5	74.7%	619	278.5	6.1%	643
ESS (8, 6)	6476524	1	438.0	68.5%	4209	1168.1	7.5%	3904
CS (4, 3)	4966	0.023	0.8	87.5%	45	2.4	8.3%	35
CS (6, 3)	34529	0.023	15.7	81.5%	214	124.1	0.9%	108
CS (6, 4)	45281	0.015	24.8	86.7%	324	243.8	0.6%	81
CS (6, 5)	56033	0.012	38.6	91.2%	312	432.1	0.4%	104
CS (7, 4)	99265	0.014	102.3	87.6%	1062	983.1	0.4%	97
CS (7, 5)	122785	0.011	161.7	92.1%	1145	1384.8	0.3%	97
CS (7, 6)	146305	0.01	245.5	94.9%	1404	2409.5	0.2%	156
CS (8, 4)	200083	0.013	585.1	93.4%	1974	-	-	-

coin protocol of the randomized consensus algorithm. CS is used as a benchmark system in the state-of-the-art probabilistic model checker PRISM [12]. Here we use a safety property in the system as our target.

The experiments based on these three models are listed in Table 2. $BSS(N)$ indicates there are N players (also N actions) in the game; $ESS(N, K)$ means there are N players and K actions; $CS(N, K)$ indicates there are N processes and K is a constant used in the model. Here we are interested in the ratio of model building (*BM*) time to the total time, which is denoted as *BMR* in the table. In $PAT(w)$, *BM* means the time for building **acyclic** DTMC, i.e., the overall time consumed by eliminating loops in DTMC; in $PAT(w/o)$, it indicates the time for building the whole system. In both PAT versions, value iteration is used to get the final result after building the model. '-' indicates the verification takes more than 1 hour thus the result is not taken into consideration. From the table, we have several observations.

1. For some small examples such as $BSS(4)$, our new approach is slower. This is due to the overhead taken by the SCC searching algorithm, and value iteration approach is efficient when loops are small.
2. As the examples become larger, the verification speed is increased by our proposed approach. This improvement is obvious especially in large-scale systems such as $ESS(8, 5)$, $ESS(8, 6)$ and $CS(8, 4)$.
3. CS consumes more resource than BSS and ESS when they have similar size of state space, such as $CS(7, 6)$ and $ESS(6, 5)$. The reason is that CS has more complicated SCCs, and both our new approach and traditional value iteration method have to use more time and memory to solve it. As a result, the SCCs' structure affects the verification efficiency to a large extent.

4. According to *BMR*, we can see that in the previous version of PAT, building the model costs small portion of the overall verification time compared with the value iteration procedure. The average value of *BMR* is less than 10%, which means slow convergence indeed exists in systems having large SCCs. CS has very small *BMR* and this is consistent with the fact that CS has complicated SCCs. In the new approach, time is mainly used by abstractions, as average *BMR* is more than 80%. It indicates that the efficiency of the divide and conquer strategy is critical in the whole verification now, and optimal dividing strategy is worthy to explore.

On the other hand, we want to share some limitations of our approach according to the experimental information. The efficiency of this approach is dependent on whether large SCCs exist in the system. During our experiment, the new approach performs slower than value iteration method in several cases. The main two reasons include 1) there is no loops in the system, thus the SCC searching algorithm makes the whole verification slow; 2) the system just has small SCCs while the whole state space is large, thus the gain of the abstraction is limited.

5 Related Work and Conclusion

SCCs are an important structure in both concurrent and probabilistic verification. For probability calculation, those loops in SCCs are one of the key factors affecting the efficiency. Some previous work has been done based on SCC decomposition for probabilistic systems, including DTMCs and Markov Decision Processes (MDPs) [5], and we are mainly inspired by this work.

To speed up the verification of MDP, the authors of [6] have proposed to decide the topological order of all SCCs in the MDP, and value iteration method is used to solve the SCCs from the bottom upwards. Based on this work, the authors of [13] have used SCC decomposition to handle the incremental quantitative verification of MDP. The topological order between SCCs guarantees that some changes in one SCC will not affect those SCCs after it. Compared to their work, ours does not consider the orders of SCCs via treating each SCC independently. This makes parallel computation approach feasible. In addition, Gaussian-Jordan elimination is used to remove loops. Different from value iteration, which needs a user defined precision, our approach generates accurate result.

Besides, there are several work based on SCC focusing on probabilistic counterexample generation, such as [3,1]. Their idea of abstracting each SCC from its input to output is the biggest inspiration of our work. Compared with these work, ours is more focusing on improving reachability analysis in DTMC. Therefore, we divide SCCs into smaller partitions and solve them directly.

Conclusion. In this work, we proposed a divide-and-conquer approach to speed up reachability analysis of DTMCs. Because SCCs are one of main reasons that the probability calculation is slow, we focus on abstracting SCCs via calculating the transition probability from their inputs to outputs. We divide every SCC, whose states exceed some specific bound, into several AGs having reasonable number of states, and can be solved efficiently via Gauss-Jordan elimination. We have implemented our approach in PAT, and some benchmark systems are used to show its effectiveness and efficiency.

For future work, there are two possible directions. Currently, the parameters used in the algorithm such as B, B_L and B_U are mainly decided via experience, and are manually defined before the experiments. Therefore, one topic is to find the more efficient division strategies, which are automatic and suitable for general cases. Another direction is extending our approach to MDP. Concurrency also exists in many probabilistic systems, so nondeterminism is unavoidable in some cases. SCCs in MDP can also be eliminated via calculating the probability distributions from inputs to outputs. Due to the nondeterminism in MDP, one challenge is that the number of resulting distributions may be exponential, thus a suitable divide and conquer approach for MDP is needed.

References

1. Ábrahám, E., Jansen, N., Wimmer, R., Katoen, J.-P., Becker, B.: DTMC Model Checking by SCC Reduction. In: QEST, pp. 37–46 (2010)
2. Althoen, S.C., McLaughlin, R.: Gauss - Jordan reduction: a brief history. The American Mathematical Monthly 94(2), 130–142 (1987)
3. Andrés, M.E., D'Argenio, P., van Rossum, P.: Significant Diagnostic Counterexamples in Probabilistic Model Checking. In: Chockler, H., Hu, A.J. (eds.) HVC 2008. LNCS, vol. 5394, pp. 129–148. Springer, Heidelberg (2009)
4. Aspnes, J., Herlihy, M.: Fast Randomized Consensus Using Shared Memory. Journal of Algorithms 15(1), 441–460 (1990)
5. Baier, C., Katoen, J.: Principles of Model Checking. The MIT Press (2008)
6. Ciesinski, F., Baier, C., Größer, M., Klein, J.: Reduction Techniques for Model Checking Markov Decision Processes. In: QEST, pp. 45–54 (2008)
7. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)
8. Grenager, T., Powers, R., Shoham, Y.: Dispersion Games: General Definitions and Some Specific Learning Results. In: AAAI, pp. 398–403 (2002)
9. Itai, A., Rodeh, M.: Symmetry Breaking in Distributed Networks. Information and Computation 88, 150–158 (1981)
10. Katoen, J.-P., Khattri, M., Zapreev, I.S.: A Markov Reward Model Checker. In: QEST, pp. 243–244 (2005)
11. Katoen, J.-P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The Ins and Outs of The Probabilistic Model Checker MRMC. In: QEST, pp. 167–176 (2009)
12. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-Time Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)
13. Kwiatkowska, M.Z., Parker, D., Qu, H.: Incremental Quantitative Verification for Markov Decision Processes. In: DSN, pp. 359–370 (2011)
14. Stoer, J., Bulirsch, R.: Introduction to Numerical Analysis. Springer, Berlin (2002)
15. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards Flexible Verification under Fairness. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 709–714. Springer, Heidelberg (2009)
16. Sun, J., Song, S., Liu, Y.: Model Checking Hierarchical Probabilistic Systems. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 388–403. Springer, Heidelberg (2010)
17. Tarjan, R.E.: Depth-First Search and Linear Graph Algorithms. SIAM J. Comput. 1(2), 146–160 (1972)
18. Younes, H.L.S., Clarke, E.M., Zuliani, P.: Statistical Verification of Probabilistic Properties with Unbounded Until. In: Davies, J. (ed.) SBMF 2010. LNCS, vol. 6527, pp. 144–160. Springer, Heidelberg (2011)

Solving Games Using Incremental Induction

Andreas Morgenstern, Manuel Gesell, and Klaus Schneider

Embedded Systems Group
Department of Computer Science
University of Kaiserslautern

Abstract. Recently, IC3 has been presented as a new algorithm for formal verification. Based on incremental induction, it is often much faster compared to otherwise used fixpoint-based model checking algorithms. In this paper, we use the idea of incremental induction for solving two-player concurrent games. While formal verification requires to prove that a given system satisfies a given specification, game solving aims at automatically synthesizing a system to satisfy the specification. This involves both universal (player 1) and existential quantification (player 2) over the formulas that represent state transitions. Hence, algorithms for solving games are usually implemented with BDD packages that offer both kinds of quantification. In this paper, we show how to compute a solution of games by using incremental induction.

1 Introduction

It is an old dream of computer science to *automatically generate* a system from a formal specification or at least to *automatically check* whether a system is guaranteed to satisfy a specification. The second problem is known as the formal verification problem and powerful tools exist to automatically check the correctness of a system with respect to a given specification. Recently, a new symbolic model checking algorithm called IC3 has been presented [4,5] that is based on incremental induction instead of the otherwise used fixpoint computations. Other researchers [6] talk about ‘Property Directed Reachability’ (PDR) in this context, since this algorithm has a very targeted approach to check the reachability of a state (violating a safety property). The newly developed algorithms often outperform existing verification engines based on bounded model checking and interpolation in practice.

The idea of synthesis or realizability [14,3,15] is to automatically construct a functionally correct system from a declarative specification. The obvious benefit is that we only have to give a list of desired behaviors and a synthesis tool comes up with a state-based model that satisfies all given properties. If no further constraints like limited use of memory or runtime requirements have to be considered, automatic synthesis can completely avoid manual coding of programs.

Synthesis can be viewed as a two-player game between an environment and a system (also called the controller). The environment chooses the uncontrollable inputs (as usual) and the controller responds by setting the controllable outputs of the system in order to satisfy the given specification. Hence, for *every* input given by the environment, the controller must choose *some* output so that the resulting game does not violate a given specification. Hence, one has to solve a quantified SAT (or QBF) problem involving a quantifier alternation. Most algorithms to solve games therefore either employ BDDs [3,16] or other data structures [7] that offer both universal and existential quantifications or replace the universal quantification by conjunctions [15] (thus blowing up the formulae).

In this paper, we propose a different solution for solving reachability games: our algorithm can be seen as a modification of PDR [6] where every SAT query has been replaced by a QBF query (more precisely, a 2QBF query). Recent results from the QBF community [10] indicate that a good way to implement a QBF solver is to use two SAT solvers; roughly speaking, one solver for existential quantification, and the other for universal quantification. For game solving, this means that the two SAT solvers take the roles of the two players. This fits nicely into our game setting, and therefore our implementation is based on ordinary SAT solvers instead of dedicated QBF solvers (although our algorithm borrows some ideas from that area). Our experiments moreover indicate that the removal of the universal quantification can be implemented efficiently if one uses the inductive generalization procedures mentioned in [4,6].

While synthesis in its most general form may be desirable, we consider here only the problem of determining the winner of a game. This is enough for two of the most promising application domains of synthesis. The first application domain is *to find errors* in an early design phase, where only a part of the whole system may be available [13]. We can now consider the problem of constructing a controller in a game that determines the outputs of the absent parts. If the controller has no winning strategy, then the already constructed parts of the system contain an error that has to be repaired before new components may be added. Another promising application for synthesis is *fault-localization* in distributed designs [11]. Typically, modern systems are composed of many different modules and determining the module that is responsible for an error is a tedious and time consuming work. Using games, one can check whether the faulty part can be replaced by a correct implementation. If this holds, chances are very high that the thereby determined candidate is actually responsible for the fault.

2 Preliminaries

A *cube* over a set of Boolean variables Q is a partial assignment of Boolean values to *some* variables in Q. We often represent cubes as a conjunction or just a set of literals (a literal is either a variable or the negation of a variable). If a cube contains *all* variables of $V_X \cup V_U \cup V_C$, it is called a *minterm*. If $d \subseteq c$ holds for a cube c, then d is called a subcube of c. A *clause* is a disjunction of literals. Given a cube $s = l_1 \wedge l_2 \wedge \cdots \wedge l_n$, its negation is a clause $\neg s = \neg l_1 \vee \neg l_2 \vee \cdots \vee \neg l_n$. We often write $\Phi(Y)$ to describe a property over the variables $Y \subseteq Q$.

A *finite state transition system* $S = (V_I, V_X, \Phi_I, \Phi_T)$ is given by a set of input variables V_I, a set of internal state variables V_X, and propositional formulas describing the initial condition $\Phi_I(V_X)$ and the transition relation $\Phi_T(V_I, V_X, V'_X)$. Given a formula $\Phi(Y)$ over a subset $Y \subseteq V_X$ of state variables, we denote with Φ' the formula that is obtained from Φ by replacing each variable x with its corresponding next-state variable x'.

A state of the system is a cube over V_X. An assignment s to all variables of a formula Φ either satisfies the formula, $s \models \Phi$, or falsifies it $s \not\models \Phi$. If s is interpreted as a state and $s \models \Phi$ holds, we say that s is a Φ-state. A formula Φ implies another formula Ψ, written $\Phi \Rightarrow \Psi$, if every satisfying assignment of Φ also satisfies Ψ. A *trace* $s_0, s_1, \ldots$ (which may have finite or infinite length) of a transition system S is a sequence of states such that $s_0 \models \Phi_I$ and for each pair s_i, s_{i+1} in the sequence, $s_i, s'_{i+1} \models \Phi_T$ holds. That is, a trace is the sequence of assignments in an execution of the transition system. A state that appears in some trace of the system is *reachable* and we denote the set of reachable states by $\mathcal{R}$. A safety property $P(V_X)$ is a propositional formula over V_X that asserts that only P-states are reachable.

3 Another Look at IC3: Computing Ranks of Fixpoints

In order to check the reachability of a bad state, i.e., one that violates a safety property, one can compute the reachable states of a system as follows: starting with the initial states, one adds successors of so-far reached states until no new states are found[1]:

$$\mathcal{R}_0 = \Phi_I$$
$$\mathcal{R}_{i+1} = \mathcal{R}_i \cup \mathsf{suc}_\exists^{\Phi_T}(\mathcal{R}_i)$$

where $\mathsf{suc}_\exists^{\Phi_T}(A) = \{s_2 \mid \exists i \in 2^{V_I}.\exists s_1 \in 2^{V_X}.\ \Phi_T(s_1, i, s'_2) \wedge s_1 \in A\}$ are the existential successors of a set of states A w.r.t. the transition relation Φ_T. Thus, for any i, $\mathcal{R}_i$ is the set of states reachable in at most i steps. Clearly, for a finite transition system, this fixpoint iteration must terminate, and there must exist a least number $\nu \in \mathbb{N}$ (called the *rank*) such that $\mathcal{R}_{\nu+1} = \mathcal{R}_\nu$ holds and the set $\mathcal{R}_\nu$ is then the set of reachable states. It is clear that a system satisfies a safety property if and only if the intersection of $\mathcal{R}_\nu$ with the states violating the safety property is empty.

Traditional BDD-based model checkers implement the above fixpoint algorithm, since BDDs are quite efficient in computing the set of all successors, but they sometimes cannot represent the transition relation Φ_T as a single BDD (and therefore consider often partitioned transition relations [17]). SAT solvers do not suffer from the latter problem. However, applying a SAT solver for computing all solutions for image computations seems to be very inefficient [8]. Nevertheless, SAT solvers played a crucial role in pushing model checkers ahead: combining the ideas of bounded model checking [1] and interpolation [12], very efficient model checkers can be implemented using SAT solvers.

[1] The computation can also be terminated if a bad state is reached by some R_i.

3.1 Applying Induction Incrementally

IC3/PDR follows a rather different way than traditional SAT-based model checkers: It can be viewed as computations of over-approximations of the reachable states. To that end, the algorithm uses incremental induction as defined below:

Definition 1 ((Inductive) Invariants). *A property $\varphi(V_X)$ is an* invariant *of a system S (i.e., an S-invariant), if $R_\nu \Rightarrow \varphi$, i.e., if only φ-states are reachable. A property $\varphi(V_X)$ is an* inductive invariant *if $\Phi_I \Rightarrow \varphi$ and $\varphi \wedge \Phi_T \Rightarrow \varphi'$.*

If P is not invariant, then there exists a finite *counterexample* trace $s_0, s_1, \ldots, s_k$ such that $s_k \not\models P$. Induction need not be applied in a monolithic way. One can construct a sequence of inductive assertions, each inductive relative to (a subset of) the previous assertions. Note that the reachable states $\mathcal{R}$ is the least inductive invariant and that there are invariants that are not inductive.

Definition 2 (Incremental Induction). *A property $\varphi(V_X)$ is* inductive relative *to another condition $\psi(V_X)$, if $\Phi_I \models \varphi$ and $\varphi \wedge \psi \wedge \Phi_T \Rightarrow \varphi'$*

PDR and IC3 use this idea of incremental induction: these algorithms incrementally refine and extend a sequence of formulas $R_0, R_1, \ldots, R_N$ that are over-approximations of the sets of states reachable in at most $0, 1, 2, \ldots, N$ steps[2]. We call this list of formulas a trace. While $R_0 = \Phi_I$ always equals the set of initial states, each R_i is represented by a set of clauses that maintains the property that $R_i \subseteq R_{i+1}$. Together with the trace, PDR maintains a set of proof obligations. A proof obligation is a cube s together with a rank k[3] where s represents a set of states that are either bad or have a trace to a bad state. The rank k can be seen as a position in a counterexample where s must be proved to be unreachable or the proof obligation fails. To obtain new informations about the trace, PDR poses the following SAT queries:

$$SAT?[R_{k-1} \wedge \Phi_T \wedge s']$$

This query holds if a state in R_{k-1} has a successor in s'. If it is not satisfiable, then the information about R_{k-1} is strong enough to show the unreachability of s within k steps. We then say that s is *blocked* at rank k, and we can add the clause $\neg s$ to R_k. Hence, s is inductive relative to R_{k-1}.

Otherwise, a new proof obligation $(s, k-1)$ is generated. If we can continue until the rank becomes 0, we have reached the initial states, and we have generated a counterexample for the safety property. If the algorithm does not succeed in generating a counterexample, at some point the informations obtained for some R_ν are strong enough to capture all reachable states. At that point, no new information is obtained and we conclude that the system is indeed safe.

[2] Hence, each $R_i \subseteq \mathcal{R}_i$, the sets calculated by the fixpoint iteration.

[3] Note that the rank is called *frame* in [6], but we prefer here rank due to the connection with the fixpoint solutions.

3.2 Model Checking by Backward Traversals

Instead of starting with the initial states and then computing new reachable states until a fixpoint is reached, one can also work backwards to verify a safety property: starting with the bad states (violating the safety property), compute the *predecessor* (instead of successor) states until no new states are found. This way, all states having a path to a bad state are finally computed. The system is safe, if and only if the initial states do not intersect with the so-computed closure of the bad states. PDR/IC3 can be modified to do the same: simply identify R_0 with the bad states; each R_i is an over-approximation of the states having a trace to bad states in at most i steps. Proof obligations (s, k) now contain a cube s representing a state known to be backwards-reachable from the bad state and the SAT query changes to $SAT?[s \wedge \Phi_T \wedge R'_{k-1}]$, i.e. checking whether for any state in R_{k-1}, there is a *predecessor* s. The solution of safety games we are going to present is based on a similar predecessor computation.

4 Games

In the following sections, we describe how IC3 can be modified to solve the following safety games:

Definition 3 (Games). *A* game $\mathcal{G} = (V_U, V_C, V_X, s_0, \Phi_T(V_X, V_U, V_C, V'_X))$ *is given by a set of uncontrollable variables V_U, a set of controllable variables V_C, a set of state variables V_X, a full cube s_0 over V_X describing the initial state of the game, and a transition relation Φ_T. The transition relation must be deterministic, i.e., for every (s, u, c), there is exactly one s' such that $(s, u, c, s') \models \Phi_T$.*

Since Φ_T is deterministic, we write in the following $\Phi_T(s, u, c) = s'$ instead of $(s, u, c, s') \models \Phi_T$ and we say that $\Phi_T(s, u, c)$ is undefined, if there is no s' such that $(s, u, c, s') \models \Phi_T$ holds. Intuitively, a game is a finite state transition system where the inputs are partitioned into controllable and uncontrollable input variables. As before, we consider a safety property $P(V_X)$ over the set of state variables. It is the goal of the controller to keep the game inside this safe region while the environment tries to reach a state where $\neg P$ holds. It is worth pointing out that safety games are determined [9], hence either the controller or the environment wins.

Given a game $\mathcal{G} = (V_U, V_C, V_X, s_0, \Phi_T(V_X, V_U, V_C, V'_X))$, a *(memoryless) strategy* for the controller is a function $\sigma : 2^{V_X} \times 2^{V_U} \to 2^{2^{V_C}}$ given by a propositional formula over V_X, V_U and V_C. Intuitively, when the game is in a state $s \in 2^{V_x}$ and the environment chooses an uncontrollable input $u \in 2^{V_u}$, a strategy determines a set of possible responses, i.e., a set $\{c_0, c_1 \dots\}$ of possible assignments $c_i \in 2^{V_c}$ to the controllable variables. A *play* on $\mathcal{G}$ according to σ is a finite or infinite sequence $\pi = s_0 \xrightarrow{u_0 c_0} s_1 \xrightarrow{u_1 c_1} \dots$ such that $c_i \in \sigma(s_i, u_i)$ and $(s_i, u_i, c_i, s_{i+1}) \models \Phi_T$. Either the play is infinite, or there is a n such that $\sigma(s_n, u_n) = \emptyset$. A play is winning according to the safety property $P(V_X)$, if it is infinite, and each $s_i \in P$. A strategy σ is a winning strategy if all plays according to σ on $\mathcal{G}$ are winning[4]. A state s is *winning* if there is a winning strategy starting in s. The set of all winning states is the winning region. The game $\mathcal{G}$ is *winning* or *won*, if the initial state s_0 is in the winning region.

It is our goal to develop an algorithm that determines whether a game is winning for the controller. If the game is winning, our algorithm will also generate a strategy for the controller. To that end, we will present in the following sections the necessary modifications to the PDR algorithm.

5 Fixpoint Computations to Solve Games

In this section, we take the viewpoint of the environment. Hence we compute the set of states from which the environment can force a visit to a $\neg P$-state, or dually, the states from which the controller loses.

Definition 4. *The set of states from which the environment can force a visit to a state in A in one step is defined as*

$$\mathsf{suc}_{\exists\forall}^{\Phi_T}(A) = \{s \in S \mid \exists u \in 2^{V_u}.\forall c \in 2^{V_c}.\forall s' \in 2^{V_{X'}}.\ \Phi_T(s,u,c,s') \rightarrow s' \in A\}.$$

A state s is in $\mathsf{suc}_{\exists\forall}^{\Phi_T}(A)$, if the environment can choose an assignment u to the uncontrollable inputs such that the controller has no chance to choose some c to prevent a visit from A. As can be seen, in contrast to the model-checking problem, we need existential (for the environment) and universal quantification (for the controller) over variables.

It is well-known that for reachability games, the winning region of the environment can be computed by the following fixpoint iteration [9]:

$$\begin{aligned}\mathcal{R}_0 &= \neg P\\ \mathcal{R}_{i+1} &= \mathcal{R}_i \cup \mathsf{suc}_{\exists\forall}^{\Phi_T}(\mathcal{R}_i)\end{aligned}$$

As for the corresponding fixpoint iteration used in model-checking, this fixpoint iteration converges to a set $\mathcal{R}_\nu$, containing the winning region of the environment. For every s in the winning region of the environment, there is a minimal n such that $s \in \mathcal{R}_n$ holds and if s is not winning, it does not belong to some R_n (and hence also not to R_ν). This leads to the following definition:

Definition 5 (Ranks of States). *The rank $\rho(s)$ of a state s with respect to the above fixpoint iteration is defined as follows:*

$$\rho(s) = \begin{cases}\top & \textit{if } \forall n \in \mathbb{N}. s \notin \mathcal{R}_n\\ \min\{n \in \mathbb{N} \mid s \in \mathcal{R}_n\} & \textit{otherwise}\end{cases}$$

[4] Note that a strategy is typically nondeterministic, but our definition ensures that we can easily select a deterministic strategy by choosing any particular assignment from 2^{V_C} for the controllable variables due to the fact that *all* plays according to σ must be winning.

Intuitively, the rank of a state s denotes how far the environment is away from reaching its goal: if $\rho(s) = \top$, the environment cannot win. Otherwise, it can drive the game to $\neg P$ in at most $\rho(s)$ steps.

6 Computing Ranks Using Incremental Induction

Our algorithm shares many similarities to the original IC3/PDR algorithm: it computes over-approximinations of ranks of states of a fixpoint formula and uses SAT queries for this purpose. Indeed, our algorithm is directly derived from the re-implementation of IC3 (called the PDR algorithm) given in [6].

6.1 Proof Obligations

In order to compute the ranks, our algorithm maintains a trace $[R_0, R_1, \dots]$, i.e., formulas representing state sets with the meaning that R_i is an over-approximation of states having rank less than i. R_0 is special: it is simply identified with the set $\neg P$.

Together with this trace, it also maintains a list of proof obligations (s, k) with the intended meaning to show that a state s has rank less than k. In order to show this, the environment must force a visit to a state with rank less than $k-1$ in one step. Hence, we have to check whether the following holds:

$$\exists u.\forall c\ .\Phi_T(s, u, c) \in R_{k-1}$$

If we cannot find such an u-value, then the facts already known in R_{k-1} are strong enough to prove that s has a rank greater than k. Hence, we remove s from R_k and we say that s is blocked at rank k.

However, if we find such a u-value, nothing can be said at that point about the rank of s since R_{k-1} only over-approximates ranks. In order to give a definitive answer, the ranks of all successor states $\{s' \mid \exists c.\ \Phi_T(s, u, c) = s'\}$ have to be probed for rank $k-1$. For every such successor state s', we therefore add a proof obligation $(s', k-1)$ to the list of proof-obligations. If we proceed this way, we might obtain a proof obligation $(s, 1)$ such that the environment can force the game into an (original) bad state in one step and prove that the game is losing for the controller. Or we strengthen some R_k (remove states from R_k) to the point where it is inductive in the sense that for every u there exists some c that is inside of R_k. In that case, the game is winning for the controller.

In order to cope with the universal quantification over the c-variables, we maintain a list of formulas $[U_0, U_1, U_2, \dots]$ over state and uncontrollable variables. The intended meaning of U_i is the following: it is an over-approximation of the set of pairs (s, u), such that every c-input leads to a R_{i+1}-state, or otherwise spoken: if we identify that $s' \notin R_{i-1}$ and for some c, we have $\Phi_t(s, u, c) = s'$, then (s, u) should be removed from U_i. Finally, we also maintain a state set W, which is an over-approximation of the states winning for the controller. Those two sets (represented as formulas) help in getting rid of the universal quantification: The query $\exists u.\forall c.\ \Phi_T(s,u,c) \in R_{k-1}$ is replaced by the query

$$SAT?[s \wedge \Phi_T \wedge R'_{k-1}]$$

If the answer is unsat, then clearly, s is blocked at rank k. Otherwise, a successor state $s' = \Phi_T(s,u,\underline{c})$ is computed for the inputs u and $\underline{c}$. Instead of continuing with the proof obligation $(s', k-1)$, the controller might give a different controllable input c with the corresponding successor $t' = \Phi_T(s,u,c)^5$. However, this successor state has to be a potential winning state. Hence, we probe

$$SAT?[s \wedge u \wedge \Phi_T \wedge W']$$

If the query is unsat, then clearly s is a losing state for the controller and we remove it from W. If s is the (losing) initial state, we can skip the rest of our calculation and terminate with the result that the game is losing for the controller. Otherwise, if the computed successor state $t' \notin R_{k-1}$, we remove (s,u) from U_i and continue with the proof obligation (s,k). Otherwise, we continue with the proof obligation $(t', k-1)$, but keep the proof obligation (s,k) in the list of open obligations. Proceeding this way, we either find that s_0 is a losing state (for the controller) or we strengthen some R_k so that it is inductive in the above sense.
The precise properties of the sets R_i, U_i, and W are:

- All R_i, U_i except R_0, U_0 are conjunctions of clauses
- $R_0 = U_0 = \neg\Phi_P$
- $R_i \Rightarrow R_{i+1}$
- The clauses of R_{i+1} are a subset of the clauses of R_i for $i > 0$.
- The clauses of U_{i+1} are a subset of the clauses of U_i for $i > 0$.
- R_{i+1} is an over-approximation of the pre-image of R_i, hence $R_{i+1} \wedge \Phi_T \Rightarrow R'_i$.
- U_{i+1} is an over-approximation of the pre-image of R_i for all c-combinations, hence, $\bigwedge_{c \in 2^{V}_{C}} (U_{i+1} \wedge c \wedge \Phi_T \Rightarrow R'_i)$.
- $R_i \Rightarrow \neg s_0$, except for the last element R_N of the trace.
- W is a conjunction of clauses that is an over-approximation of the winning positions for the controller. Hence $W \Rightarrow P$.

6.2 Notation

Let Φ be a predicate over the game variables, let Ψ be a predicate over (next)-state variables and let Φ_T denote the encoding of the transition relation. Given cubes s_0, u_0 over state and uncontrollable input variables, a call to the underlying SAT solver will be expressed similarly as in [6]:

$$(isSat, s, u, c, t') \leftarrow SAT?[s_0 \wedge u_0 \wedge \Phi \wedge \Phi_T \wedge \Psi']$$

5 Due to efficiency reasons, this probing for an alternative successor is only done if s' is a state outside of W, i.e., a state known to be losing for the controller.

This query asks whether the environment can choose an uncontrollable input u_0 in a state where s_0 and (a formula) Φ holds, so that a state where Ψ holds is reached in one step, i.e., can the system choose a controllable assignment to make the game reach a state where Ψ holds?

The answer to this question is put into the Boolean variable $isSat$. If the answer is positive, the satisfying assignment is put into (s, u, c, t') with the obvious meaning: s denotes the assignment to the state variables, u to the uncontrollable variables, c to the controllable variables, and t' to the next-state variables. Modern SAT solvers not only compute a solution to SAT problems in case of success, but also produce reasons for a failed SAT call. If the aforementioned SAT-call fails, we assume that the SAT-solver computes subcubes $s \subseteq s_0$ and $u \subseteq u_0$ of the given assumptions s_0 and u_0 (t' contains no value in that case).

6.3 Auxiliary Functions

In order to present our algorithm, we need some auxiliary functions that are used to update the sets R_i, U_i and W. Note that the only updates to one of those sets is the removal of states which can be readily implemented using cubes and clauses: Given a cube s representing a set of states or transitions, the clause $\neg s$ represents all states, resp. transitions outside of s. Hence, the implementation of the following auxiliary functions are straightforward[6]:

- addLose(s) adds s as a losing state, i.e., updates $W \leftarrow W \wedge \neg s$
- addBlockedState(s, k) updates $R_k \leftarrow R_k \wedge \neg s$. Due to the syntactic containment restriction, we have to update also $R_i \leftarrow R_i \wedge \neg s$ for every $0 \leq i \leq k$.
- addBlockedTransition(s, u, k) updates $U_i \leftarrow U_i \wedge \neg s$ for every $0 \leq i \leq k$.
- isLose(s) checks whether s is a losing state
- isBlocked(s, k) checks whether s is blocked at rank k

6.4 Recursively Blocking Cubes

In this section, we discuss the function recBlockCube given in Listing 1.1. Given a proof obligation (s_0, k_0), this function checks whether the rank of s_0 is greater than k_0, i.e., if s_0 is blocked at rank k_0. The main internal data structure of this function is a priority queue Q that stores open proof obligations that are needed to decide (s_0, k_0). The following lemma states that the invariants about R_i, U_i, and W are maintained by our algorithm:

Lemma 1. *If (s, k) is the minimal element of Q and the invariants of R_i, U_i and W hold, then one of the following situations may occur:*

1. *s is correctly identified as blocking at rank k*

[6] For an efficient implementation of the algorithm, it is important that the functions addLose, addBlockedState and addBlockedTransition learn strong facts, meaning small clauses. For addBlockedState, we use the same literal removal procedure as given in [6] while for addBlockedTransition, we use ternary simulation minimization, also described in [6].

2. *s is correctly identified as losing*
3. *For some u, (s,u) is correctly identified as a blocking transition at rank k and (s,k) is again added as a proof obligation.*
4. *$k > 1$, (s,k) and $(t',k-1)$ are added to Q for some successor t' of s*

Listing 1.1. Recursively Blocking a Cube

```
1         recBlockCube( ProofObligation ($s_0, k_0$) ) {
2            PrioQ<ProofObligation> Q; – order from low to high ranks
3            $Q.add((s,k))$;
4            while (Q.size()>0){
5               $(s,k) \leftarrow Q.popMin()$;
6               if(isLose($s$))
7                     if($s == s_0$) return isLose
8               else if(not isBlocked(s,k)) {
9                     $(isSat, s, u, c, t') \leftarrow SAT?[s \wedge U_k \wedge \Phi_T \wedge R'_{k-1}]$
10              if($isSat$){
11                    if(($k == 1$)$or$(isLose($t'$))){
12                       $(isSat, s, u, c, t') \leftarrow SAT?[s \wedge u \wedge \Phi_T \wedge W']$
13                       if($isSat$){
14                          if(($k == 1$)$or$(isBlocked($t', k-1$)))
15                             addBlockedTransition(s,u,k);
16                          else
17                             $Q.add(t', k-1, Some(s,u,c))$;
18                          $Q.add(s,k,pre)$;
19                       }
20                       else
21                          addLose(s) ;
22                          if($s == s_0$) return isLose;
23                    }
24                    else
25                       $Q.add(t', k-1, Some(s,u,c))$
26                       $Q.add(s,k,pre)$
27                    }
28                    else
29                 addBlockedState(s,k);
30              }
31           }
32              return ISGREATER
```

Proof. Let us first consider the case $k = 1$. In that case, $R_{k-1} = R_0 = \neg\Phi_P$, hence R_{k-1} represents the bad states. That means that if the SAT query in line 9: $SAT?[s \wedge U_k \wedge \Phi_T \wedge R'_{k-1}]$ yields 'not satisfiable', then s is correctly identified as blocking in line 29. Otherwise, if it yields 'satisfiable', then state s has a successor in the bad states, hence in the next line, isLose is true . If the controller cannot avoid a losing position in line 12, $SAT?[s \wedge u \wedge \Phi_T \wedge W']$, then clearly s is a losing position which is identified in line 21. Otherwise, (s,u) is correctly identified as a blocking transition in line 15. Hence, for $k = 1$, one of the first three cases occurs. Now assume that $k > 1$ holds. Clearly, if the checks in line 6 or line 8 succeed, either case 1 or 2 applies. Otherwise, the algorithm proceeds to line 9, and we can make the following case distinctions referring to the situations of the above lemma:

- Case 1 occurs, if the SAT query in line 9 yields ‘not satisfiable’. Correctness follows from the invariance of R_i and U_i.
- Case 2 occurs, if the SAT query in line 9 yields ‘satisfiable’, t' is a losing position (line 11) and the SAT query in line 12 yields ‘not satisfiable’. Correctness follows from the invariance of R_i (for query in line 12) and W (for query in line 12).
- Case 3 occurs, if the SAT query in line 9 yields ‘satisfiable’, t' is a losing position (line 11), and the SAT query in line 12 yields ‘satisfiable’. Correctness follows from the invariance of R_i, since we have identified for (s, u) some c such that the successor t' is blocked at rank $k - 1$.
- Case 4 can occur if the SAT query in line 9 yields ‘satisfiable’ and if one of the following cases occur:
  - t' is a losing position (line 11) and the SAT query in line 12 yields ‘not satisfiable’
  - t' is no losing position (line 11)

□

The correctness of the algorithm is stated in the following theorem:

Theorem 1. *Given a proof obligation* (s_0, k_0) *for some* $k_0 > 0$*, the function* recBlockCube(s_0, k_0) *returns ISGREATER, if the rank of* s *is greater than* k *and isLose, if* s_0 *is a losing position for the controller. Function* recBlockCube *moreover updates* R_k*,* U_k *and* W *such that this new information is stored, but keeps the invariants of all sets* R_i, U_i *and* W*.*

Proof. If the function returns through lines 7 or 22, we know that the invariants of the sets are kept, so that s_0 is a losing position. Now note that the following holds: Q cannot grow arbitrarily: We can prove by induction on k that the following holds: If (s, k) is chosen as the minimal element of Q, then one of the first three cases of the previous lemma are encountered after a finite number of steps. The base case $k = 1$ is already handled by the previous lemma. For the induction step $k \to k + 1$, note that if we have got a proof obligation $(s, k + 1)$, then s has only a finite number of successor states t' that may be added as a proof obligation (t', k) to Q. For all (t', k), we can apply the induction hypothesis. Moreover, if we have processed all successor states (computed the rank or identified some of them as losing), we can determine the rank of s or show that s is losing. If the procedure returns with ISGREATER at the end, previously Q must be emptied. Now note that the following holds: whenever (s_0, k_0) is chosen as the minimal element, it is either identified as losing or as blocking, or added again as a proof obligation. Hence, if the while-loop is left, the if-condition in line 8 must fail. □

6.5 Main Function

Our main function is given in Listing 1.2. It first checks whether the initial state is a bad state. If so, it returns FALSE. Otherwise, our internal data structures are initialized. It then recursively probes for $k = 1 \ldots$ whether the rank of the

Listing 1.2. Main Function

```
bool main() {
  $(isSat, \_, \_, \_, \_) \leftarrow SAT?[s_0 \wedge \neg P]$
  if ($isSat$) return FALSE;
  $R_0 \leftarrow \neg P$
  $R_i \leftarrow 1$,  for all $i > 0$ – meaning: $R_i = 2^{V_X}$
  $U_0 \leftarrow \neg P$ for all $i>0$
  $U_i 1$,  for all $i > 0$ – meaning : $U_i \leftarrow 2^{(V_X \cup V_U)}$
  clauses($L$) $\leftarrow \emptyset$,  for all $i > 0$ – meaning : $L \leftarrow 2^{V_X}$
  for (k=1 .. ) do {
    if (recBlockCube(($s_0, k$) == $isLose$)) return FALSE
    propagateBlockedStates(k);
    if (clauses($R_i$)=clauses($R_{i+1}$)) for $1 \leq i < k$ return TRUE
  }
}
```

initial state equals k. If this is the case, then clearly the controller loses the game. Otherwise, the function propagateBlockedStates is called: if a state s was identified as blocked at rank i, but it is also blocked at rank $i+1$, then the corresponding clause $\neg s$ is also added to R_{i+1}. Finally, as in the original IC3 [4] or PDR algorithm [6], if we find that some adjacent levels R_i and R_{i+1} share all clauses, then R_i is an inductive strengthening of $\neg s_0$, hence the initial state is not backwards reachable from the bad states. This is captured by the following theorem:

Theorem 2. *The function main given in listing 1.2 computes a solution of the game: it returns TRUE if and only if the controller has a winning strategy.*

Proof (sketch). Clearly, if the procedure returns FALSE, due to the correctness of recBlockCube, we have shown that the initial state has a finite rank and hence is a losing position. Otherwise, if it returns TRUE, then for some k, we have $R_k = R_{k+1}$ and since R_k is an over-approximation of the set computed in the fixpoint iteration, the game is indeed safe. Finally, can k grow infinitely? Clearly, if the check in the last line of main would be done semantically, then this clearly could not happen. R_{k+1} would have to block at least one state less than R_k. Suppose therefore that $R_k = R_{k+1}$ holds, but $clauses(R_k) \neg clauses(R_{k+1})$. During the propagation phase in propagateBlockedStates, all clauses of R_k will be moved into R_{k+1} and they become syntactically equivalent. □

7 Experiments

We have implemented a prototype of our algorithm, called IC3G, in Microsoft's new language F# with an interface to Minisat 2.2 and evaluated different case studies. We have also implemented a safety game algorithm with an interface to the popular BDD-package CUDD in our framework. Unfortunately, the latter performed so poorly[7] that we decided to rather compare with a tool from the

[7] It could solve only the smallest benchmarks AMBA2 and GenBuf2 with a space limit of 2 GB, while the SAT-algorithm uses only a few MB.

Listing 1.3. Propagating Blocked States

```
void propagateBlockedStates() {
for i = 1 .. k
    for each $\neg s \in$ clauses($R_i$) - $s$ is blocked at rank $k$
        ($isSat$, _, _, _, _) $\leftarrow$ $SAT?[s \wedge R'_{i+1} \wedge U_{i+1} \wedge \Phi_T]$
 if (not isSat) {
        - $s$ is also blocked at rank $k+1$
        $R_{i+1} = R_{i+1} \wedge \neg s$
 }
}
```

literature. We therefore use the tool Marduk [2] which is a BDD-based implementation of the algorithm described in [3] for so-called GR(1)-specifications[8]. It is implemented in Python with an interface to the BDD-package CUDD.

The first case study is the GenBuf example which consists of a family of buffers. The task is to generate a controller that handles in/output for those buffers. The second example is *ARM's Advanced Microcontroller Bus Architecture (AMBA)* which defines the *Advanced High performance Bus (AHB)*, an on-chip communication standard that connects devices like processor cores, caches and DMA arbiters. Here, we want to synthesize an arbiter for the bus. Both case studies can be seen as standard benchmarks that have been used before to evaluate game solving algorithms [3,7,16] and can be parametrized by a parameter that represents the number of clients served.

In [3], temporal logic specifications as well as deterministic ω-automata are given for these benchmarks. The games we build are obtained as the automaton product of the deterministic automata, hence they contain fairness constraints. To obtain a safety-game, we use a (simplified) version of the bounded approach to synthesis described in [7,15].

Model	Marduk	IC3G	Model	Marduk	IC3G
GenBuf 2	0.08	0.5	Amba2	0.7	0.9
GenBuf 4	0.15	1.3	Amba4	2.8	1.6
GenBuf8	1.22	2.5	Amba8	43.1	2.7
GenBuf16	1.68	4.1	Amba16	92.5	7.2

Fig. 1. Experimental Results: Running time in seconds for computing the winner

All experiments have been run on an Intel Core 2 Duo with 2.66 Ghz, the IC3G algorithm under Windows 7 and Marduk under Ubuntu Linux. The results of our experiments are summarized in Figure 1 where we have listed the runtimes in seconds of our tool IC3G and Marduk[9]. On the GenBuf example, that can be solved by both algorithms in a couple of seconds, our tool is slightly slower

[8] GR(1) specifications have the form $\bigwedge_i \varphi_i \to \bigwedge_j \psi_j$ with fairness constraints φ_i, ψ_j.

[9] The runtime for Marduk only contains the time needed for strategy generation; the output function generation is not counted.

than the BDD-based algorithm. This changes when we consider the AMBA case study, which contains much more state variables than the GenBuf example. Here, our algorithm is significantly faster. This is no surprise, since on big examples with many variables, BDDs often suffer from memory requirements so that they can no longer be efficiently handled.

The experiments we performed do however only consider one part of game solving: we only compute the winner of a game, but we have not looked at the problem of actually computing a winning strategy. Computing the winning region of the controller is the first step in generating a winning strategy in each algorithm for controller synthesis. Known (BDD-based) algorithms can be easily adjusted to compute from the winning region a winning strategy. For safety-games one can obtain a simple winning strategy from controller's winning region: all we have to do is to forbid every transition that leads from a winning position to a non-winning position. However, since we only compute an over-approximation for the winning states, our algorithm is not able to construct a winning strategy in that straightforward way. However, we expect that it is possible to modify our algorithm for that purpose so that we can also obtain a winning strategy using incremental induction. This would then solve also the last application domain we sketched for game solving: automatically constructing a system from a temporal logic specification.

8 Conclusions

In the past, many improvements have been suggested to increase the performance of model checking tools. Starting with symbolic model checking based on BDDs, bounded model checking based on SAT solvers was used, and then interpolation-based model checking even allowed to use SAT solvers for unbounded model checking. Recently, incremental induction has been proposed as an alternative to the so-far used fixpoint-based methods and it turned out to be much more efficient for model checking. Controller synthesis or equivalent problems like game solving are similar to model checking, but have to face the additional problem of alternating quantifiers which is no problem for BDD-based approaches, but requires QBF solvers instead of SAT solvers otherwise. In this paper, we have shown how we can use a simple SAT solver for game solving by following the ideas of the recently introduced incremental induction procedures, and we experienced similar improvements concerning the efficiency of our tools. While the experiments are still quite preliminary, they indicate that incremental induction may be as useful for game solving as for model-checking.

References

1. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
2. Bloem, R., Cimatti, A., Greimel, K., Hofferek, G., Könighofer, R., Roveri, M., Schuppan, V., Seeber, R.: RATSY – A new requirements analysis tool with synthesis. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 425–429. Springer, Heidelberg (2010)
3. Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Specify, compile, run: Hardware from PSL. Electronic Notes in Theoretical Computer Science (ENTCS), vol. 190, pp. 3–16 (2007)
4. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
5. Bradley, A.R.: IC3 and beyond: Incremental, inductive verification. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 4–4. Springer, Heidelberg (2012)
6. Eén, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: Bjesse, P., Slobodová, A. (eds.) Formal Methods in Computer-Aided Design (FMCAD), pp. 125–134. IEEE Computer Society, Austin (2011)
7. Filiot, E., Jin, N., Raskin, J.-F.: Compositional algorithms for LTL synthesis. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 112–127. Springer, Heidelberg (2010)
8. Grumberg, O., Schuster, A., Yadgar, A.: Memory efficient all-solutions SAT solver and its application for reachability analysis. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 275–289. Springer, Heidelberg (2004)
9. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata, Logics, and Infinite Games. LNCS, vol. 2500. Springer, Heidelberg (2002)
10. Janota, M., Marques-Silva, J.: Abstraction-based algorithm for 2QBF. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 230–244. Springer, Heidelberg (2011)
11. Jobstmann, B.: Applications and Optimizations for LTL Synthesis. PhD thesis, IST – Institute for Software Technology, TU Graz, Graz, Austria (February 2007)
12. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
13. Nopper, T., Scholl, C.: Approximate symbolic model checking for incomplete designs. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 290–305. Springer, Heidelberg (2004)
14. Rosner, R.: Modular Synthesis of Reactive Systems. PhD thesis, The Weizmann Institute of Science, Israel, Rehovot, Israel (1992)
15. Schewe, S., Finkbeiner, B.: Bounded synthesis. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 474–488. Springer, Heidelberg (2007)
16. Sohail, S., Somenzi, F., Ravi, K.: A hybrid algorithm for LTL games. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 309–323. Springer, Heidelberg (2008)
17. Somenzi, F.: Binary decision diagrams. In: Broy, M., Steinbrüggen, R. (eds.) *Calculational System Design*. NATO Science Series F: Computer and Systems Sciences, vol. 173, pp. 303–366. IOS Press (1999)

Model-Checking Software Library API Usage Rules[⋆]

Fu Song and Tayssir Touili

LIAFA, CNRS and Univ. Paris Diderot, France
{song,touili}@liafa.univ-paris-diderot.fr

Abstract. Modern software increasingly relies on using libraries which are accessed via Application Programming Interfaces (APIs). Libraries usually impose constraints on how API functions can be used (API usage rules) and programmers have to obey these API usage rules. However, API usage rules often are not well-documented or documented informally. In this work, we show how to use the SCTPL logic to precisely specify API usage rules in libraries, where SCTPL can be seen as an extension of the branching-time temporal logic CTL with variables, quantifiers, and predicates over the stack. This allows library providers to formally describe API usage rules without knowing how their libraries will be used by programmers. We also propose an approach to automatically check whether programs using libraries violate or not the corresponding API usage rules. Our approach consists in modeling programs as pushdown systems (PDSs), and checking API usage rules on programs using SCTPL model checking for PDSs. To make the model-checking procedure more efficient, we propose an abstraction that reduces drastically the size of the program model. Moreover, we characterize a sub-logic rSCTPL of SCTPL preserved by the abstraction. rSCTPL is sufficient to precisely specify all the API usage rules we met. We implemented our techniques in a tool and applied it to check several API usage rules. Our tool detected several previously unknown errors in well-known programs, such as Nssl, Verbs, Acacia+, Walksat and Getafix. Our experimental results are encouraging.

1 Introduction

Most modern software increasingly relies on using libraries and frameworks provided by organizations in order to shorten time to market. Libraries or frameworks are accessed via Application Programming Interfaces (APIs) which are sets of library functions (called API functions) and usually impose constraints (API usage rules) on how API functions can be used. Programmers have to obey these constraints when calling API functions. However, most of API usage rules are not well-documented or documented informally in the API documentation. It is easy to introduce bugs using API functions. So, it is important to formally describe and automatically check API usage rules.

Many works addressed this problem [15, 19, 22, 24–26, 28, 30, 32–36, 38, 44, 45, 47]. However, their approaches either cannot describe API usage rules in a precise manner or cannot automatically check API usage rules. In this paper, we propose a novel technique to specify and check API usage rules without knowing how API functions will be

[⋆] Work partially funded by ANR grant ANR-08-SEGI-006.

used by programmers. Our approach consists of (1) modeling programs as pushdown systems (PDSs), since PDSs are a natural model of sequential programs [23] (the stack of PDSs stores the calling procedures which allows us to check context-sensitive API usage rules), (2) specifying in a precise manner API usage rules in the Stack Computation Tree Predicate Logic (SCTPL) [41] (indeed, SCTPL can describe several API usage rules that cannot be expressed by the existing works), and (3) automatically checking whether programs violate or not API usage rules by SCTPL model checking for PDSs.

SCTPL can be seen as an extension of the CTPL logic with predicates over the stack content. CTPL [29] is an extension of the Computation Tree Logic (CTL) with variables and quantifiers. In CTPL, propositions can be predicates of the form $p(x_1, ..., x_m)$, where the x_i's are free variables or constants. Free variables can get their values from a finite domain and be universally or existentially quantified. CTPL can specify API usage rules without knowing how API functions will be used by programmers. E.g., consider the file operation API usage rule "The file should be closed by calling the API function *fclose* whenever this file is opened by calling *fopen*". Closing opened files is important. Indeed, long time running programs, such as web servers, will use many resources if opened files are not closed. This API usage rule can be expressed in CTL as $\psi_1 \equiv \mathbf{AG}(fopen \Longrightarrow \mathbf{EF}fclose)$ (note that the formula $\psi_1' \equiv \mathbf{AG}(fopen \Longrightarrow \mathbf{AF}fclose)$ is incorrect, if *fopen* returns a *null* file pointer, then *fclose* should not be called). However, ψ_1 cannot detect the bug in Figure 2, where the file pointed to by f_2 is never closed. This is due to the fact that we cannot specify the relation between the return value of *fopen* and the parameter of *fclose*. To detect this bug, we have to specify this rule as $\psi_2 \equiv \mathbf{AG}(\bigwedge_{i=1}^{3}(f_i = fopen \Longrightarrow \mathbf{EF}fclose(f_i)))$. However, this formula is too special to specify this rule in library, since e.g., replacing the variable f_1 by f_1' breaks ψ_2. Using CTPL, we can specify this rule as $\psi_3 \equiv \forall x \forall y \forall z\, \mathbf{AG}(x = fopen(y, z) \Longrightarrow \mathbf{EF}fclose(x))$ stating that whenever a file is opened and pointed to by some variable x, it should be closed in the future. [1]

```
n1 : FILE* f1=fopen("t1","w");
n2 : FILE* f2=fopen("t2","w");
n3 : FILE* f3=fopen("t3","w");
n4 : if(f1) then
n5 :   fclose(f1);
n6 : fclose(f3);
```

Fig. 2. File Operations

However, CTPL cannot specify properties about the calling procedures. Being able to express such properties is important. E.g., consider an API usage rule expressing that "Calling a function $proc_1$ in some procedure $proc$ must be followed by a call to the function $proc_2$ before the procedure $proc$ returns". This API usage rule cannot be specified in CTPL. To overcome this problem, we use the SCTPL logic [40, 41] to precisely describe API usage rules. SCTPL extends CTPL by predicates over the stack. Such predicates are given by regular expressions over the stack alphabet and some free

[1] Note that ψ_3 cannot express the point that *fclose* is only called when *fopen* returns a pointer to the file. Indeed, *fopen* returns a null pointer when the file does not exist. In this case, calling *fclose(f_3)* induces an error. To express such a point, we introduce an additional predicate *Test(x)* which holds at some control point n iff x is tested at the control point n. Now, we can refine the rule into $\psi_4 \equiv \forall x \forall y \forall z\, \mathbf{AG}(x = fopen(y, z) \Longrightarrow \mathbf{AF}(Test(x) \wedge \mathbf{EXAF}fclose(x)))$. ψ_4 states that whenever $x = fopen$ is made, one has to check the return value x (i.e., *Test(x)*). After this, the file has to be closed in all the future paths. The motivation of using *Test(x)* is that we cannot know how the return value will be checked. Thus, we coarsely specify that the return value is checked.

variables (which can also be existentially and universally quantified). Using SCTPL, the above rule can be specified as $\forall l\ \mathbf{AG}((proc_1 \wedge \Gamma l \Gamma^*) \Longrightarrow \mathbf{AF}(proc_2 \wedge \Gamma^+ l \Gamma^*))$, where $\Gamma l \Gamma^*$ and $\Gamma^+ l \Gamma^*$ are regular predicates. The subformula $(proc_1 \wedge \Gamma l \Gamma^*)$ expresses that $proc_1$ is called inside some procedure *proc* whose return address is l (since the return addresses of the called procedures are put into the stack when executing the program.). The above formula states that whenever $proc_1$ is called in some procedure *proc* whose return address is l (ensured by $\Gamma l \Gamma^*$), a function call to $proc_2$ should be made where the return address l is still in the stack, i.e., before the procedure *proc* returns (this is ensured by $\Gamma^+ l \Gamma^*$). Note that, in our modeling, the topmost symbol of the stack of the PDS stores the current control point, the rest of the stack stores the return addresses of the calling procedures, i.e., the procedures that have not returned yet.

It is shown in [41] that SCTPL model checking for PDSs is decidable. Thus, we can automatically check whether a program violates or not API usage rules by SCTPL model-checking for PDSs. To make the verification of API usage rules more efficient, we introduce the *procedure-cutting abstraction*, which is an abstraction that drastically reduces the size of the program model by removing some procedures that do not use the API functions specified in the SCTPL formula. We also consider rSCTPL, a sub-logic of SCTPL and show that the procedure-cutting abstraction preserves all rSCTPL formulas when the removed procedures are infinite execution free. rSCTPL is sufficient to express all the API usage rules we met. Moreover, rSCTPL can describe all API usage rules we met. Our abstraction allowed us to apply our techniques to large programs.

The main contributions of this paper are:

1. We propose a novel approach to precisely specify API usage rules using SCTPL. SCTPL allows library providers to formally describe API usage rules when implementing the libraries.
2. We can automatically check programs against API usage rules by SCTPL model-checking. Our techniques also allow program developers to automatically verify API usage rules of their programs without any additional inputs nor environment abstractions.
3. We propose a procedure-cutting abstraction. We show that this abstraction preserves all rSCTPL formulas when the cut procedures are infinite execution free. Our abstraction reduces drastically the size of the program model, which makes API usage rules verification more efficient.
4. We implemented our techniques in a tool and applied it to check several API usage rules on several open source programs. Our tool was able to find several unknown bugs in some well-known open source programs, such as Nssl, Verbs, Acacia+, Walksat and Getafix.

Outline. Section 2 gives a formal definition of PDSs. Section 3 recalls the definition of SCTPL, and shows how to precisely specify API usage rules in SCTPL. Section 4 describes the procedure-cutting abstraction and the sub-logic rSCTPL of SCTPL. Section 5 discusses the experimental results. The related work is given in Section 6.

2 Formal Model: Pushdown Systems

In this section, we recall the definition of pushdown systems. We use the approach of [23] to model a sequential program as a pushdown system.

A *Pushdown System* (PDS) is a tuple $\mathcal{P} = (P, \Gamma, \Delta)$, where P is a finite set of control locations, Γ is the stack alphabet, $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of transition rules. A configuration $\langle p, \omega \rangle$ of $\mathcal{P}$ is an element of $P \times \Gamma^*$. We write $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$ instead of $((p, \gamma), (q, \omega)) \in \Delta$. The successor relation $\rightsquigarrow_{\mathcal{P}} \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$ is defined as follows: if $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$, then $\langle p, \gamma\omega' \rangle \rightsquigarrow_{\mathcal{P}} \langle q, \omega\omega' \rangle$ for every $\omega' \in \Gamma^*$. A *path* of the PDS is a sequence of configurations $c_1 c_2 ...$ such that c_{i+1} is an *immediate successor* of the configuration c_i, i.e., $c_i \rightsquigarrow_{\mathcal{P}} c_{i+1}$, for every $i \geq 1$.

3 API Usage Rules Specification

In this section, we recall the definition of the Stack Computation Tree Predicate Logic (SCTPL) [41], and show how to specify API usage rules in SCTPL.

3.1 Environments, Predicates and Regular Variable Expressions

Hereafter, we fix the following notations. Let $\mathcal{X} = \{x_1, x_2, ...\}$ be a finite set of variables ranging over a finite domain $\mathcal{D}$. Let $B : \mathcal{X} \cup \mathcal{D} \longrightarrow \mathcal{D}$ be an environment function that assigns a value $v \in \mathcal{D}$ to each variable $x \in \mathcal{X}$ and such that $B(v) = v$ for every $v \in \mathcal{D}$. $B[x \leftarrow v]$ denotes the environment function such that $B[x \leftarrow v](x) = v$ and $B[x \leftarrow v](y) = B(y)$ for every $y \neq x$. Let $\mathcal{B}$ be the set of all the environment functions.

Let AP be a finite set of atomic propositions, $AP_{\mathcal{X}}$ be a finite set of atomic predicates in the form of $a(\alpha_1, ..., \alpha_m)$ such that $a \in AP$, $\alpha_i \in \mathcal{X} \cup \mathcal{D}$ for every $1 \leq i \leq m$, and $AP_{\mathcal{D}}$ be a finite set of atomic predicates of the form $a(\alpha_1, ..., \alpha_m)$ such that $a \in AP$, $\alpha_i \in \mathcal{D}$ for every $1 \leq i \leq m$.

Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, let $\mathcal{R}$ be a finite set of regular variable expressions over $\mathcal{X} \cup \Gamma$ given by: $e ::= \emptyset \mid \epsilon \mid a \in \mathcal{X} \cup \Gamma \mid e + e \mid e \cdot e \mid e^*$.

The language $L(e)$ of a regular variable expression e is a subset of $P \times \Gamma^* \times \mathcal{B}$ defined inductively as follows: $L(\emptyset) = \emptyset$; $L(\epsilon) = \{(\langle p, \epsilon \rangle, B) \mid p \in P, B \in \mathcal{B}\}$; $L(x)$, where $x \in \mathcal{X}$ is the set $\{(\langle p, \gamma \rangle, B) \mid p \in P, \gamma \in \Gamma, B \in \mathcal{B} : B(x) = \gamma\}$; $L(\gamma)$, where $\gamma \in \Gamma$ is the set $\{(\langle p, \gamma \rangle, B) \mid p \in P, B \in \mathcal{B}\}$; $L(e_1 + e_2) = L(e_1) \cup L(e_2)$; $L(e_1 \cdot e_2) = \{(\langle p, \omega_1\omega_2 \rangle, B) \mid (\langle p, \omega_1 \rangle, B) \in L(e_1);\ (\langle p, \omega_2 \rangle, B) \in L(e_2)\}$; and $L(e^*) = \{(\langle p, \omega \rangle, B) \mid B \in \mathcal{B}$ and $\omega = \omega_1 \cdots \omega_m$, s.t. $\forall i, 1 \leq i \leq m, (\langle p, \omega_i \rangle, B) \in L(e)\}$. E.g., $(\langle p, \gamma_1\gamma_2\gamma_2 \rangle, B)$ is an element of $L(\gamma_1 x^*)$ when $B(x) = \gamma_2$.

3.2 Stack Computation Tree Predicate Logic

A SCTPL formula is a CTL formula where predicates and regular variable expressions are used as atomic propositions and variables can be quantified. Regular variable expressions are used to express predicates on the stack content of the PDS. More precisely, the set of *SCTPL formulas* is given by (where $x \in \mathcal{X}$, $a(x_1, ..., x_m) \in AP_{\mathcal{X}}$ and $e \in \mathcal{R}$):

$$\varphi ::= a(x_1, ..., x_m) \mid e \mid \neg\varphi \mid \varphi \wedge \varphi \mid \forall x\, \varphi \mid \mathbf{EX}\varphi \mid \mathbf{EG}\varphi \mid \mathbf{E}[\varphi \mathbf{U} \varphi].$$

Let φ be a SCTPL formula. The closure $cl(\varphi)$ denotes the set of all the subformulas of φ including φ.

Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$ s.t. $\Gamma \subseteq \mathcal{D}$, let $\lambda : AP_{\mathcal{D}} \rightarrow 2^{\Gamma}$ be a labeling function that assigns a set of stack symbols to a predicate. Let $c \in P \times \Gamma^*$ be a configuration of $\mathcal{P}$. $\mathcal{P}$ satisfies a SCTPL formula ψ in c, denoted by $c \models_\lambda \psi$, iff there exists an environment $B \in \mathcal{B}$ s.t. $c \models_\lambda^B \psi$, where $c \models_\lambda^B \psi$ is defined by induction as follows:

- $c \models_\lambda^B a(x_1, ..., x_m)$ iff $n \in \lambda(a(B(x_1), ..., B(x_m)))$ and $c = \langle p, n\omega \rangle$.
- $c \models_\lambda^B e$ iff $(c, B) \in L(e)$.
- $c \models_\lambda^B \psi_1 \wedge \psi_2$ iff $c \models_\lambda^B \psi_1$ and $c \models_\lambda^B \psi_2$.
- $c \models_\lambda^B \forall x\, \psi$ iff $\forall v \in \mathcal{D},\ c \models_\lambda^{B[x \leftarrow v]} \psi$.
- $c \models_\lambda^B \neg\psi$ iff $c \not\models_\lambda^B \psi$.
- $c \models_\lambda^B \mathbf{EX}\, \psi$ iff there exists a successor c' of c s.t. $c' \models_\lambda^B \psi$.
- $c \models_\lambda^B \mathbf{E}[\psi_1 \mathbf{U} \psi_2]$ iff there exists a path $\pi = c_0 c_1 ...$ of $\mathcal{P}$ with $c_0 = c$ s.t. $\exists i \geq 0,\ c_i \models_\lambda^B \psi_2$ and $\forall 0 \leq j < i, c_j \models_\lambda^B \psi_1$.
- $c \models_\lambda^B \mathbf{EG}\psi$ iff there exists a path $\pi = c_0 c_1 ...$ of $\mathcal{P}$ with $c_0 = c$ s.t. $\forall i \geq 0$: $c_i \models_\lambda^B \psi$.

Intuitively, $c \models_\lambda^B \psi$ holds iff the configuration c satisfies ψ under the environment B. We will freely use the following abbreviations: $\mathbf{AX}\psi = \neg\mathbf{EX}(\neg\psi)$, $\mathbf{EF}\psi = \mathbf{E}[true\mathbf{U}\psi]$, $\mathbf{AG}\psi = \neg\mathbf{EF}(\neg\psi)$, $\mathbf{AF}\psi = \neg\mathbf{EG}(\neg\psi)$, $\mathbf{A}[\psi_1\mathbf{U}\psi_2] = \neg\mathbf{E}[\neg\psi_2\mathbf{U}(\neg\psi_1 \wedge \neg\psi_2)] \wedge \neg\mathbf{EG}\neg\psi_2$, $\mathbf{A}[\psi_1\mathbf{R}\psi_2] = \neg\mathbf{E}[\neg\psi_1\mathbf{U}\neg\psi_2]$, $\mathbf{E}[\psi_1\mathbf{R}\psi_2] = \neg\mathbf{A}[\neg\psi_1\mathbf{U}\neg\psi_2]$, and $\exists x\psi = \neg\forall x\neg\psi$.

Theorem 1. *[41] SCTPL model-checking for PDSs is decidable.*

3.3 Extracting Predicates for API Specifications

API usage rules often state properties concerning the order of API function calls and return value tests. Indeed, usually, after making a call to an API function, one has to check whether the call was successful. For example, when *fopen* is called to open a file *t1*, one has to make sure that the call was successful, i.e., that the file *t1* exists (as done in Figure 2, Line n_4). Thus, to check API usage rules, we need to extract predicates about API function calls and return value tests. To do this, for every API function call $y = f(p_1, ..., p_m)$ at a control point n where y denotes the return value[2] and for every $1 \leq i \leq m$, p_i denotes the i^{th} parameter of the function f, we add the predicate $f(p_1, ..., p_m, y)$ to $AP_{\mathcal{D}}$ and associate this predicate to the control point n (i.e., we let $n \in \lambda(f(p_1, ..., p_m, y))$). By abuse of notation, such predicates $f(p_1, ..., p_m, y)$ will also be denoted by $y = f(p_1, ..., p_m)$.

For every boolean expression b in a conditional statement (e.g., if-then-else, switch-case) at a control point n s.t. y is used in b and y is a return value of some function call, we add the predicate *Test(y)* in $AP_{\mathcal{D}}$ and associate this predicate to n (i.e., we let $n \in \lambda(Test(y))$).

Intuitively, for every $\omega \in \Gamma^*$, a configuration $\langle s_0, n\omega \rangle$ satisfies the atomic predicate σ (i.e., σ is $y = f(p_1, ..., p_m)$ or *Test(x)*) iff σ is associated to n (i.e., $n \in \lambda(\sigma)$). W.l.o.g., we suppose that the return value of some API function is immediately checked in the same procedure where the API function is called. This assumption will not restrict the usefulness of the libraries, and it is recommended to check the return value immediately after the function call.

[2] W.l.o.g., we assume that each function call has a return value assigned to some variable.

$\lambda(f_1 = fopen(``t_1", ``w")) = \{n_1\}$	$\langle s_0, n_1\rangle \hookrightarrow \langle s_0, fo_0 n_2\rangle$
$\lambda(f_2 = fopen(``t_2", ``w")) = \{n_2\}$	$\langle s_0, n_2\rangle \hookrightarrow \langle s_0, fo_0 n_3\rangle$
$\lambda(f_3 = fopen(``t_3", ``w")) = \{n_3\}$	$\langle s_0, n_3\rangle \hookrightarrow \langle s_0, fo_0 n_4\rangle$
$\lambda(Test(f_1)) = \{n_4\}$	$\langle s_0, n_4\rangle \hookrightarrow \langle s_0, n_5\rangle$
$\lambda(fclose(f_1)) = \{n_5\}$	$\langle s_0, n_5\rangle \hookrightarrow \langle s_0, fc_0 n_6\rangle$
$\lambda(fclose(f_3)) = \{n_6\}$	$\langle s_0, n_6\rangle \hookrightarrow \langle s_0, fc_0 n_7\rangle$
(a)	(b)

Fig. 2. (a) The labeling function λ and (b) Transition rules Δ

3.4 An Illustrating Example

To illustrate our approach, we show how to specify the API usage rules for the GNU socket library.

Description of the Socket Library The socket library implements a generalized interprocess communication channel. It provides TCP and UDP Protocols. As shown in Figure 3, a server-side program using the TCP Protocol should first create a socket *s* by calling *socket* with SOCK_STREAM as second parameter, then bind *s* to some address by calling *bind* and listen to the address by calling *listen*. When the server receives a connection request, it will create a new socket *ns* by calling *accept*. Then, the server can communicate with the client by calling *send* and *recv* via the socket *ns*. Finally, *s* and *ns* should be destroyed by calling *close*.

Figure 4 shows a typical application of the TCP Protocol at the client-side. It connects to a server by calling *connect* after creating the socket *s*. Then, it can communicate with the server by calling *send* and *recv* via the socket *s*. Finally, *s* should be destroyed by calling *close*.

The server-side program using the UDP Protocol should create a socket *s* by calling *socket* with SOCK_DGRAM as second parameter as shown in Figure 5. After that, it should bind *s* to some address by calling *bind*. Then, it can communicate with a client by calling *recvfrom* and *sendto* via *s*. Finally, the socket *s* should be closed by calling *close*. The client-side program using the UDP Protocol can communicate with a server by calling *recvfrom* and *sendto* via a socket *s* after its creation. Figure 6 is a typical implementation of the UDP Protocol at the client-side.

```
int s, c, ns;
if((s=socket(AF_INET,SOCK_STREAM,0))== -1)
  return;
if(bind(s,&s_addr,len)==-1)
  {close(s); return;}
if(listen(s,5)==-1){close(s); return;}
while(1){
  ns=accept(s,&c_addr, &size);
  do{
    recv(ns,data,256,0);
    ...
    send(ns,data,256,0);
    if(cond1){close(ns); return;}
  }while(cond2)
}
close(s);
```

Fig. 3. TCP Server-side

```
int s;
if((s=socket(AF_INET,SOCK_STREAM,
              0))==-1)
  return;
...
connect(s,&s_addr,len)
do{
  send(s,data,256,0);
  ...
  recv(s,data,256,0);
}while(cond3)
close(s);
```

Fig. 4. TCP Client-side

```
int s;
if((s=socket(AF_INET, SOCK_DGRAM,0))==-1)
  return;
if(bind(s,&s_addr,sizeof(s_addr))==-1)
  { close(s); return; }
do{
  recvfrom(s,data,256,0,&c_addr, len);
  sendto(s,data,256,0,&c_addr,len);
}while(cond4)
close(s);
```

Fig. 5. UDP Server-side

```
int s;
if((s=socket(AF_INET,SOCK_DGRAM,
            0))==-1)
  return;
do(1){
  sendto(s,data,256,0,&addr, len);
  ...
  recvfrom(s,data,256,0,&addr, len);
 }while(cond5)
close(s);
```

Fig. 6. UDP Client-side

Specifying the Socket Library API Usage Rules in SCTPL. Table 1 shows some SCTPL formulas describing some API usage rules of the socket library. Let us consider the API usage rule "The return value of *socket* should be checked immediately after the call to *socket* is made, and after a socket is created, this socket should be destroyed in all the future paths". We can specify this rule by the SCTPL formula r_1 as shown in Table 1. r_1 states that whenever the call to *socket* is made in a procedure *proc* whose return address is l (the regular predicate $\Gamma l \Gamma^*$ ensures that the return address of the procedure *proc* is l), the return value stored in the variable y should be eventually checked in all the future paths (i.e., *Test(y)*) inside this procedure (this is ensured by the fact that the stack is still of the form $\Gamma l \Gamma^*$ when the test of y is made). After this test, the socket y should be eventually closed in all the future paths (this is ensured by **EXAF** $close(y)$). The other rules in Table 1 are explained as follows.

The formula r_2 states that whenever *bind* is called to bind the socket to some address in a procedure whose return address is l, the user has to check whether the binding is correct before this procedure returns. r_3 and r_4 are similar to r_2.

The formula r_5 specifies that a socket y should be created ($y = socket(-,-,-)$) prior to binding the socket y to some address ($bind(y,-,-)$), where $-$ matches any constant (i.e., a variable quantified by $\forall$). r_6 is similar to r_5.

The formula r_7 states that any occurrence of $connect(y,-)$ should be preceded by an occurrence of $y = socket(-, SOCK\_STREAM, -)$ using the TCP Protocol.

Table 1. A set of API usage rules of the Socket Library extracted from the Socket library manual

No.	Rule
r_1	$\forall y\, \forall l\, \mathbf{AG}\, \big((y = socket(-,-,-) \wedge \Gamma l \Gamma^*) \Longrightarrow \mathbf{AF}\, (Test(y) \wedge \Gamma l \Gamma^* \wedge \mathbf{EX}\, \mathbf{AF}\, close(y))\big)$
r_2	$\forall y\, \forall l\, \mathbf{AG}\, (y = bind(-,-,-) \wedge \Gamma l \Gamma^* \Longrightarrow \mathbf{AF}\, (Test(y) \wedge \Gamma l \Gamma^*))$
r_3	$\forall y\, \forall l\, \mathbf{AG}\, (y = listen(-,-) \wedge \Gamma l \Gamma^* \Longrightarrow \mathbf{AF}\, (Test(y) \wedge \Gamma l \Gamma^*))$
r_4	$\forall y\, \forall l\, \mathbf{AG}\, (y = connect(-,-,-) \wedge \Gamma l \Gamma^* \Longrightarrow \mathbf{AF}\, (Test(y) \wedge \Gamma l \Gamma^*))$
r_5	$\forall y\, \mathbf{A}[y = socket(-,-,-)\, \mathbf{R}\, \neg bind(y,-,-)]$
r_6	$\forall y\, \mathbf{A}[listen(y,-)\, \mathbf{R}\, \neg accept(y,-,-)]$
r_7	$\forall y\, \mathbf{A}[y = socket(-, SOCK\_STREAM, -)\, \mathbf{R}\, \neg connect(y,-,-)]$
r_8	$\forall y\, \mathbf{A}[(y = socket(-, SOCK\_STREAM, -) \wedge \mathbf{A}[bind(y,-,-)\, \mathbf{R}\, \neg listen(y,-)])\, \mathbf{R}\, \neg listen(y,-)]$
r_9	$\forall y\, \mathbf{A}[connect(y,-,-) \vee y = accept(-,-,-)\, \mathbf{R}\, \neg send(y,-,-,-)]$
r_{10}	$\forall y\, \mathbf{A}[connect(y,-,-) \vee y = accept(-,-,-)\, \mathbf{R}\, \neg recv(y,-,-,-)]$
r_{11}	$\mathbf{AG}\, \forall y\, (y = accept(-,-,-) \Longrightarrow \mathbf{AF}\, close(y))$
r_{12}	$\forall y\, \mathbf{A}[y = socket(-, SOCK\_DGRAM, -)\, \mathbf{R}\, \neg(sendto(y,-,-,-,-,-) \vee recvfrom(y,-,-,-,-,-))]$
r_{13}	$\forall y\, \mathbf{A}[sendto(y,-,-,-,-,-) \vee bind(y,-,-)\, \mathbf{R}\, \neg recvfrom(y,-,-,-,-,-)]$

The formula r_8 specifies that any occurrence of listening to a socket y ($listen(y, -)$) should be preceded by an occurrence of creating the socket y using the TCP Protocol ($y = socket(-, SOCK\_STREAM, -)$), and the socket y should be bound to some address ($bind(y, -, -)$) before listening.

The formula r_9 states that before sending a data ($send(y, -, -, -)$) via a socket y, the socket y should either be connected to the target server at the client-side ($connect$ $(y, -, -)$) or y should be the socket created by $y = accept(-, -, -)$ at the server-side. r_{10} is similar.

The formula r_{11} specifies that the new socket created by $y = accept(-, -, -)$ should be eventually closed ($close(y)$) in all the future paths.

The formula r_{12} states that the socket should be created using the UDP Protocol ($y = socket(-, SOCK\_DGRAM, -)$) prior to sending ($sendto(y, -, -, -, -)$) or receiving ($recvfrom$ $(y, -, -, -, -)$) some data using the UDP Protocol.

The formula r_{13} specifies that before receiving ($recvfrom(y, -, -, -, -)$) some data using the UDP Protocol, one has to send some data ($sendto(y, -, -, -, -)$) to the server at the client-side or bind ($bind(y, -, -)$) the socket to some address at the server-side. Since using the UDP protocol, no connection is created, the client sends data by specifying the target address in the third parameter of the function *sendto*. After this, the client can receive data from the server. The server can send data only after receiving the client address from some client.

Checking the API Usage Rules. Consider the program in Figure 3. If *cond1* is true (Fig. 3: line 13), the socket s will never be closed. The SCTPL formula r_1 can detect this bug by model-checking the program against r_1. Consider the program in Figure 4, if the client managed to connect to a server which only supports the UDP Protocol as in Figure 5, the connection at line 5 of Figure 4 will fail, then sending (Figure 4: line 7) or receiving (Figure 4: line 9) some data via the socket s will induce an error. This error can be detected by checking the SCTPL formula r_4.

4 rSCTPL and The Procedure-Cutting Abstraction

To make API usage rules verification more efficient, it is important to model programs by PDSs having *small size*. We propose in this section to use the *procedure-cutting abstraction* to drastically reduce the size of the program model. The procedure-cutting abstraction removes all the procedures whose runs don't call any API function specified in the given SCTPL formula. We characterize a sub-logic rSCTPL of SCTPL that is sufficient to specify all the API usage rules that we met, and we show that the procedure-cutting abstraction preserves all rSCTPL formulas.

4.1 Procedure-Cutting Abstraction

Let $\mathcal{M}$ be a program that consists of a finite set of procedures $Proc = \{proc_i \mid 1 \leq i \leq m\}$. Each procedure $proc_i$ will generate transition rules in the PDS model. Imagine there exists some procedure $proc_j$ whose runs do not call any API function specified in the given SCTPL formula ψ, then removing $proc_j$ will not change the satisfiability of

ψ. This means that the procedure $proc_j$ can be cut. Cutting such procedure $proc_j$ will drastically reduce the size of the PDS model. We call this *procedure-cutting abstraction*. From the PDS's point of view, a function call statement $y = proc_j(...)$ at a control point n (suppose n' is the next control point of n) is represented by the transition rule $\rho = \langle s_0, n\rangle \hookrightarrow \langle s_0, e_{proc_j} n'\rangle$ where e_{proc_j} denotes the entry control point of the procedure $proc_j$. Whenever the procedure $proc_j$ can be cut, we will add the transition rule $\rho' = \langle s_0, n\rangle \hookrightarrow \langle s_0, n'\rangle$ instead of ρ. The transition rule ρ' expresses that the run from n will immediately move to n' without entering the procedure $proc_j$. By doing the procedure-cutting abstraction, the size of the stack alphabet and transition rules will be drastically reduced.

Formally, to compute the abstracted program, we proceed as follows. Let $\mathcal{M}$ be a program, a *call graph* of $\mathcal{M}$ is a tuple $G = (Proc, E, proc_0)$, where $Proc$ is a finite set of nodes denoting the procedure names of $\mathcal{M}$; $E \subseteq Proc \times Proc$ is a finite set of edges such that $(proc_i, proc_j) \in E$, denoted by $proc_i \longrightarrow proc_j$, iff $proc_j$ is called in the procedure $proc_i$; $proc_0 \in Proc$ is the initial node corresponding to the entry procedure (usually, the *main* function) of $\mathcal{M}$. A node $proc_i$ can reach the node $proc_j$ iff there exists a set of edges $proc_{k_1} \longrightarrow proc_{k_2}, ..., proc_{k_m} \longrightarrow proc_{k_{m+1}}$ in E such that $k_1 = i$ and $k_{m+1} = j$. Let $Op(\psi) = \{proc \in AP \mid \exists proc(x_1, ..., x_m) \in cl(\psi) \wedge proc \neq Test\}$ denote the set of atomic propositions (i.e., API function names) used in the SCTPL formula ψ except the additional atomic proposition *Test*. The procedure-cutting abstraction computes the abstracted program $\mathcal{M}'$ by (1) removing all the procedures $proc \in Proc$ s.t. the node $proc$ cannot reach any node of $Op(\psi)$ in G (i.e., the run of $proc$ will not call any function in $Op(\psi)$), and (2) replacing each function call $y = proc(p_1, ..., p_m)$ by a *skip* statement, i.e., no operation statement.

Proposition 1. *Given a program $\mathcal{M}$ and a SCTPL formula ψ, we can compute the abstracted program $\mathcal{M}'$ in linear time.*

4.2 The rSCTPL Logic

The procedure-cutting abstraction can drastically reduce the size of the program model. However, it cannot preserve all SCTPL formulas. Indeed, formulas using the **X** operator without any restriction are not preserved, since the procedure-cutting abstraction removes procedures in the programs and replaces some function calls by *skip*. However, formulas of the form $a(x_1, ..., x_m) \wedge \mathbf{EX}\phi$ and $a(x_1, ..., x_m) \wedge \mathbf{AX}\phi$ are preserved when ϕ is a regular predicate e or its negation $\neg e$ or a SCTPL formula using the **X** operator as in the above form. Indeed, if the predicate $a(x_1, ..., x_m)$ appearing in a SCTPL formula (a function call or a return value test) is made in some procedure $proc$, then all the procedures including $proc$ whose runs can reach $proc$ will not be removed by the procedure-cutting abstraction. This implies that the next control point of $a(x_1, ..., x_m)$ will not be removed and the stack content at the next control point in the abstracted program $\mathcal{M}'$ is the same as in $\mathcal{M}$.

Moreover, formulas using regular variable expressions (e.g. e, $\neg e$) without any restriction are not preserved. Indeed, control points in $\mathcal{M}$ satisfying e or $\neg e$ may be removed by the procedure-cutting abstraction. Thus, the runs of $\mathcal{M}'$ cannot reach these control points. However, formulas of the form $a(x_1, ..., x_m) \wedge e$ or $a(x_1, ..., x_m) \wedge \neg e$

are preserved. Since all the procedures which can reach the procedure *proc* where $a(x_1, ..., x_m)$ is made are not removed, each control point in $\mathcal{M}$ satisfying $a(x_1, ..., x_m)$ has the same calling procedures (i.e., stack content) as in $\mathcal{M}'$. Then, a configuration of $\mathcal{M}$ satisfies $a(x_1, ..., x_m) \wedge e$ iff this configuration of $\mathcal{M}'$ satisfies $a(x_1, ..., x_m) \wedge e$.

Based on the above observations, we define rSCTPL as follows (where $a(x_1, ..., x_m) \in AP_{\mathcal{X}}$, $x \in \mathcal{X}$, and $e \in \mathcal{R}$):

$$\begin{aligned}\varphi ::= {} & a(x_1, ..., x_m) \mid \neg a(x_1, ..., x_m) \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \forall x\, \varphi \mid \exists x\, \varphi \\ & \mid \mathbf{A}[\varphi \mathbf{U} \varphi] \mid \mathbf{E}[\varphi \mathbf{U} \varphi] \mid \mathbf{A}[\varphi \mathbf{R} \varphi] \mid \mathbf{E}[\varphi \mathbf{R} \varphi] \mid a(x_1, ..., x_m) \wedge \psi \\ \psi ::= {} & e \mid \neg e \mid \mathbf{EX} e \mid \mathbf{AX} e \mid \mathbf{EX}\neg e \mid \mathbf{AX}\neg e \mid \mathbf{EX}\varphi \mid \mathbf{AX}\varphi\end{aligned}$$

Intuitively, rSCTPL is a sub-logic of SCTPL, where (1) the next time operator $\mathbf{X}$ is used only to specify that a rSCTPL formula ψ or a regular predicate e or its negation $\neg e$ holds immediately after an atomic predicate holds (i.e., an API function call is made or a return value is tested), and (2) regular predicates and their negations are conjuncted with atomic predicates. rSCTPL is sufficient to specify all the API usage rules we met.

However, the procedure-cutting abstraction does not preserve rSCTPL formulas when a cut procedure has an infinite execution. For instance, let $n_1 \xrightarrow{stmt} n_2$ be an edge s.t. *stmt* is a function call $y = f(p_1, ..., p_m)$ and the procedure f has an infinite execution. Suppose we replace this function call by *skip*. If n_1 and all the control locations of f don't satisfy the atomic predicate a (i.e., API function calls or return value test), while n_2 satisfies a, then the configuration $\langle s_0, n_1\omega\rangle$ of $\mathcal{M}$ satisfies $\mathbf{EG}\neg a$, but $\langle s_0, n_1\omega\rangle$ does not satisfy $\mathbf{EG}\neg a$ in $\mathcal{M}'$ due to the removal of the infinite execution. On the other hand, if n_1 and all the control locations of f do not satisfy the atomic predicate a, while n_2 satisfies the atomic predicate b, then the configuration $\langle s_0, n_1\omega\rangle$ of $\mathcal{M}'$ satisfies $\mathbf{A}[\neg a \mathbf{U} b]$ due to the removal of the infinite execution, while $\langle s_0, n_1\omega\rangle$ does not satisfy $\mathbf{A}[\neg a \mathbf{U} b]$ in $\mathcal{M}$ (since b is never true in the infinite execution). We can show the following theorem.

Theorem 2. *Let ψ be a rSCTPL formula. Let $\mathcal{M}$ be a program and $\mathcal{M}'$ be the program obtained from $\mathcal{M}$ by applying the procedure-cutting abstraction. Let $\mathcal{P}$ (resp. $\mathcal{P}'$) be the PDS modeling the program $\mathcal{M}$ (resp. $\mathcal{M}'$). If all the removed procedures are infinite execution free, then $\mathcal{P}$ satisfies ψ iff $\mathcal{P}'$ satisfies ψ.*

5 Experiments

We implemented our techniques in a tool for API usage rules verification. Given a program $\mathcal{M}$ using some libraries which are equipped with the API usage rules specified in rSCTPL, our tool automatically answers either Yes or No, depending on whether the program violates the API usage rules or not.

In our implementation, we use goto-cc [31] to parse ANSI-C programs into goto-cc binary programs. We implemented a translator translating goto-cc binary programs into pushdown systems and outputs the required predicates as discussed in Section 3.3. We use the SCTPL model-checker of [41] as engine. In our experiments, we consider several API usage rules: the socket library API usage rules and the file operation usage rules. We checked several open-source C programs against these API usage rules. All the experiments were run on a Linux platform (Fedora 13) with a 2.4GHz CPU and 2GB of memory. The time limit is fixed to 30 minutes. Our tool detected several previously unknown errors in some well-known open source programs. The run time consists of the time spent for parsing goto-cc binary programs and model-checking. It excludes the time for translating ANSI-C programs into goto-cc binary programs. We also run our tool without considering the procedure-cutting abstraction. We observed that the procedure-cutting abstraction significantly speeds up the analysis.

5.1 Checking The Socket Library API Usage Rules

To check the socket library API usage rules shown in Table 1, we checked seven open-source programs from SourceForge [12] which are written in C and use the socket library, and four generic tutorial socket programs written by Seshadri [37].

The benchmark contains the following programs. **Comserial** is a program that helps turn console application into a web based service, by reading from TCP connections and providing commands from each connection to applications through a socket. **Mr-ChaTTY** is a chat program that allows users to chat via UNIX terminals through sockets. **Mrhttpd** is a web server. **Nerv** is a common socket server. **Nssl** is a netcat-like program with SSL support. **Pop3client** is a mail client which reads mail in a console and connects to servers using POP3 Protocol. **Ser2nets** is a program allowing network connections to remote serial ports. **TCPC**, **TCPS**, **UDPC** and **UDPS** are a TCP client, a TCP server, a UDP client and a UDP server tutorial programs, respectively.

Table 2 shows the results of checking the socket library API usage rules with the procedure-cutting abstraction. The row *#LOC* gives the number of lines of the program. For $1 \leq i \leq 13$, the row r_i depicts the results of checking the API usage rule r_i against these programs, where the rows $Time(s)$ and $Mem(MB)$ give the time consumption in seconds and memory consumption in MB, respectively. The result *Proved* denotes that the program satisfies the corresponding API usage rule, *FA* denotes *false alarm* and *Bug* denotes a real bug. *o.o.m.* (resp. *o.o.t.*) means run out of memory (resp. time).

We can see from Table 2, there are 22 alarms including *Bug* and *FA*. We found that 12 of these alarms are real bugs and the others are false alarms. These false alarms arose from the fact that we abstract away the data. We found 12 real errors in these programs. For instance, the program **Comserial** does not call *listen* before calling *accept* in the file *passwdserver.c* when *argc* is 1. Moveover, most of these programs will not close the socket by calling *close* nor check the return values of *socket* in some paths. E.g., **Comserial** does not check the return value (i.e., socket) in the file *comserver.c* before it is used. In the file *main.c*, when it fails in binding a socket to some address, **Mrhttpd** will not close this socket before the program terminates.

5.2 Checking File Operation Usage Rules

File reading and writing are frequently used in programs. To read or write a file, a user has to correctly open the file by calling *fopen* which returns a file pointer to the file. Then the user can read from or write to that file. Finally the file pointer should be closed by calling *fclose*.

For file operation API usage rules, we consider two rules from *stdio.h*: $F_1 = \mathbf{AG}\, \forall\, y \big(y = fopen(-,-) \Longrightarrow \mathbf{AF}(Test(y) \wedge \mathbf{EXAF} fclose(y))\big)$ and $F_2 = \forall\, y\, \mathbf{A}[y = fopen(-,-)$

Table 2. Results of checking the socket library API usage rules with the procedure-cutting abstraction

Program		Comserial	MrChaTTY	Mrhttpd	Nerv	Nssl	Pop3client	Ser2nets	TCPC	TCPS	UDPC	UDPS
#LOC		1.0k	1.2k	1.4k	1.1k	1.1k	1.6k	7.3k	70	90	50	60
r_1	Time(s)	0.08	0.26	0.29	7.94	1.24	0.41	70.53	0.01	0.01	0.01	0.01
	Mem(MB)	0.24	0.44	0.66	5.94	1.44	0.58	11.63	0.09	0.13	0.06	0.06
	Result	Bug	FA	Bug	FA	Bug	Bug	Bug	Bug	Bug	Bug	Bug
r_2	Time(s)	0.01	0.09	0.01	0.04	0.23	0.01	8.72	0.01	0.01	0.01	0.01
	Mem(MB)	0.06	0.35	0.07	0.24	0.36	0.01	2.04	0.01	0.01	0.01	0.01
	Result	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved
r_3	Time(s)	0.01	0.09	0.01	0.03	0.11	0.01	9.57	0.01	0.02	0.01	0.01
	Mem(MB)	0.05	0.37	0.07	0.20	0.29	0.01	2.03	0.01	0.10	0.01	0.01
	Result	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved
r_4	Time(s)	0.01	0.01	0.01	0.11	0.16	0.09	6.31	0.01	0.01	0.01	0.01
	Mem(MB)	0.01	0.01	0.01	0.29	0.33	0.29	1.72	0.07	0.01	0.01	0.01
	Result	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved
r_5	Time(s)	0.01	0.01	0.01	0.01	0.01	0.01	0.18	0.01	0.01	0.01	0.01
	Mem(MB)	0.04	0.18	0.05	0.22	0.19	0.14	1.07	0.04	0.06	0.04	0.04
	Result	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved
r_6	Time(s)	0.06	0.01	0.01	0.01	0.01	0.01	0.20	0.01	0.03	0.01	0.01
	Mem(MB)	0.15	0.18	0.05	0.19	0.01	0.01	1.12	0.01	0.10	0.01	0.01
	Result	Bug	Proved	Proved	Proved	FA	Proved	Proved	Proved	Proved	Proved	Proved
r_7	Time(s)	0.01	0.01	0.01	0.02	0.02	0.02	0.21	0.01	0.01	0.01	0.01
	Mem(MB)	0.04	0.15	0.05	0.22	0.19	0.18	0.92	0.05	0.05	0.04	0.04
	Result	Proved	Proved	Proved	Proved	Bug	FA	Proved	Proved	Proved	Proved	Proved
r_8	Time(s)	0.01	0.07	0.01	0.09	0.07	0.03	1.03	0.01	0.01	0.01	0.01
	Mem(MB)	0.07	0.47	0.08	0.54	0.44	0.30	2.86	0.07	0.12	0.05	0.05
	Result	Proved	Proved	Proved	Proved	FA	Proved	Proved	Proved	Proved	Proved	Proved
r_9	Time(s)	0.01	0.01	0.01	0.02	0.01	0.01	0.07	0.02	0.01	0.01	0.01
	Mem(MB)	0.11	0.34	0.30	0.50	0.29	0.30	1.46	0.08	0.10	0.01	0.01
	Result	Proved	FA	Proved	Proved	Proved	FA	Proved	Proved	Proved	Proved	Proved
r_{10}	Time(s)	0.01	0.01	0.01	0.05	0.01	0.01	0.07	0.01	0.01	0.01	0.01
	Mem(MB)	0.11	0.33	0.33	0.75	0.29	0.35	1.46	0.08	0.09	0.01	0.01
	Result	Proved	FA	Proved	FA	Proved	FA	Proved	Proved	Proved	Proved	Proved
r_{11}	Time(s)	0.10	0.56	0.32	-	-	0.13	-	0.02	0.03	0.01	0.01
	Mem(MB)	0.47	1.97	1.50	o.o.m.	o.o.m.	0.39	o.o.m.	0.11	0.17	0.01	0.01
	Result	Bug	Proved	Proved	-	-	Proved	-	Proved	Proved	Proved	Proved
r_{12}	Time(s)	0.01	0.01	0.01	0.01	0.01	0.01	0.04	0.01	0.01	0.01	0.01
	Mem(MB)	0.04	0.15	0.05	0.18	0.15	0.14	0.71	0.04	0.05	0.05	0.04
	Result	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved
r_{13}	Time(s)	0.01	0.01	0.01	0.01	0.01	0.01	0.07	0.01	0.03	0.01	0.01
	Mem(MB)	0.05	0.31	0.07	0.17	0.30	0.01	1.46	0.01	0.10	0.05	0.05
	Result	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved

R$\neg(fread(-,-,-,y) \vee fwrite(-,-,-,y))$]. F_1 states that whenever a file is opened by calling *fopen* where y stores its return file pointer (i.e., $y = fopen(-,-)$), we need to check whether the opening of the file is correct (i.e., *Test(y)*), and there exists a next point after checking y such that the file is eventually closed (i.e., **EX AF** $fclose(y)$). F_2 states that the user cannot read from or write to a file pointer y unless the file pointer y points to some file (i.e., has already been opened).

To evaluate these two rules, we checked the following open source programs which use file API functions from *stdio.h*. **Verbs** is a bounded model checker [10]. **Getafix** is a symbolic model-checker for recursive boolean programs [3]. **Moped** is a model-checker for pushdown systems [7]. **Acacia+** is a tool for LTL realizability and synthesis [1]. **Mist** is a solver of the coverability problem for monotonic extensions of Petri nets [6]. **Elastic** is a translator from elastic specifications to hytech or uppaal language [2]. **Mckit**

Table 3. Results of checking the API usage rules F_1 and F_2 with the procedure-cutting abstraction

Program		Verbs	Getafix	Moped	Acacia+	Mist	Elastic	Mckit	TSPASS	MiniSat	Walksat	Ubcsat
#LOC		4.0k	11.5k	30.3k	8.0k	16.0k	15.4k	26.7k	62.3k	1.4k	1.4k	16.9k
F_1	Time(s)	0.96	0.18	9.92	0.05	0.01	1.45	-	0.25	0.01	0.06	216.88
	Mem(MB)	1.17	0.36	10.52	0.20	0.10	2.99	o.o.m.	0.67	0.08	0.28	15.92
	Result	Bug	Bug	Proved	Bug	Proved	Proved	-	Proved	Proved	Bug	FA
F_2	Time(s)	0.08	0.29	9.67	0.01	0.26	0.89	23.60	0.01	0.01	0.01	0.06
	Mem(MB)	0.50	0.84	10.26	0.09	0.90	2.94	15.00	0.27	0.27	0.13	0.89
	Result	FA	Proved	FA	Proved	FA	FA	Proved	Proved	FA	Proved	Proved

is a model-checking Kit [4]. **TSPASS** is a fair automated theorem prover for monodic first-order temporal logic with expanding domain semantics and propositional linear-time temporal logic [8]. **Walksat**, **MiniSat** and **Ubcsat** are three SAT solvers [5, 9, 11].

Table 3 shows the results of checking these programs against F_1 and F_2 with the procedure-cutting abstraction. As shown in Table 3, we found that **Verbs**, **Getafix**, **Acacia+** and **MiniSat** have real errors. E.g., in the file *main.c*, **Verbs** does not close an opened file by calling *fclose* before the program terminates. Moreover, in the files *issat.c*, *main.c* and *util.c*, a file pointer is used without checking whether it is *NULL* or not (i.e., whether the file exists or not). **Acacia+**, **Walksat** and **Getafix** do not close opened files which are opened in *main.c*, *walksat.c*, *bpsuspend.y* and *bp.y*, respectively.

6 Related Work

There has been a lot of works on API usage rules specification and checking [13–16, 19, 22, 24–26, 28, 30, 32–36, 38, 44–47]. However, all these works cannot specify context-sensitive specifications, whereas our approach can.

Some tools dedicated to software model-checking were used to check API usage rules for device drivers, such as DDVerify [46]. But, these tools can only check safety properties. Other works on software model-checking, such as [17, 18, 27, 42, 43], could be applied to check API usage rules. However, all these works cannot check full CTL properties.

Model-checking is used to verify security-critical applications in which security vulnerabilities are expressed by safety properties over API functions [20, 21]. However, these works consider only safety properties.

Code contracts introduced in [24] can specify pre/post-conditions and invariants for each API function. Programmers have to make sure that a pre-condition (resp. post-condition) holds at the entry (resp. exit) of each API function, and that invariants always hold inside the API function. These code contracts can be verified via either runtime checking or static checking at compile time. However, they cannot specify relations between API functions which are often used in API usage rules.

Mining-based methods are proposed [13–15, 19, 22, 25, 26, 30, 32, 33, 35, 38, 45, 47] to discover API usage rules from executing traces or source codes, where API usage rules are represented by some patterns or finite automata. One can apply model-checking techniques to check whether programs violate or not API usage rules represented by patterns or finite automata. However, all these works cannot specify data dependencies between API functions' parameters and return values of API functions. This disallows one to precisely express API usage rules. Variables are introduced into finite automata to specify data dependencies between API functions in [15, 28]. However, these works cannot express CTL-like properties (e.g., the above file operation API usage rule), and do not show how to check whether programs violate or not API usage rules represented by finite automata equipped with variables.

A class of temporal properties, called QBEC, is used to specify API usage rules using at most one temporal operator [34]. We can show that SCTPL is more expressive than QBEC. Indeed, all the temporal operators in QBEC can be expressed by SCTPL formulas. Ramanathan et al propose a formalism in [36] to specify data-dependence between API functions. However, they only consider mining preconditions of API functions rather than verification. CTL extended with variables is proposed to specify API usage rules in [44]. However, this work cannot specify context-sensitive specifications which is important for API usage rules.

SCTPL is introduced in our previous work [41], in which SCTPL is used to express malicious behaviors and model-checking is applied to detect malware. Although, SCTPL is as expressive as CTL with regular valuations [39], in [41], we have shown that SCTPL model-checking is more efficient than CTL model-checking with regular valuations.

References

1. Acacia+, `http://lit2.ulb.ac.be/acaciaplus/`
2. elastic, `http://www.ulb.ac.be/di/ssd/madewulf/aasap/`
3. Getafix, `http://www.cs.uiuc.edu/madhu/getafix/`
4. Mckit, `http://www.fmi.uni-stuttgart.de/szs/tools/mckit/`
5. Minisat, C.: language version, `http://minisat.se/MiniSat.html`
6. Mist2, `http://software.imdea.org/pierreganty/software.html`
7. Moped, `http://www.fmi.uni-stuttgart.de/szs/tools/moped/`
8. Tspass, `http://www.csc.liv.ac.uk/michel/software/tspass/`
9. Ubcsat, `http://ubcsat.dtompkins.com/`
10. Verbs, `http://lcs.ios.ac.cn/zwh/verbs/index.html`
11. Walksat, version 35, `http://www.cs.rochester.edu/kautz/walksat/`
12. SourceForge (2012), `http://sourceforge.net`
13. Acharya, M., Xie, T.: Mining API error-handling specifications from source code. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 370–384. Springer, Heidelberg (2009)
14. Acharya, M., Xie, T., Pei, J., Xu, J.: Mining API patterns as partial orders from source code: From usage scenarios to specifications. In: ESEC/FSE 2007 (2007)
15. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: POPL (2002)
16. Besson, F., Jensen, T.P., Métayer, D.L.: Model checking security properties of control flow graphs. Journal of Computer Security (2001)
17. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker BLAST. In: STTT (2007)
18. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. IEEE Trans. Software Eng. 30(6) (2004)
19. Chen, F., Roşu, G.: Mining parametric state-based specifications from executions. Technical report (2008)

20. Chen, H., Dean, D., Wagner, D.: Model checking one million lines of C code. In: NDSS (2004)
21. Chen, H., Wagner, D.: Mops: an infrastructure for examining security properties of software. In: ACM Conference on Computer and Communications Security (2002)
22. Dallmeier, V., Lindig, C., Wasylkowski, A., Zeller, A.: Mining object behavior with ADABU. In: WODA (2006)
23. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithm for model checking pushdown systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, Springer, Heidelberg (2000)
24. 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)
25. Gabel, M., Su, Z.: Javert: fully automatic mining of general temporal properties from dynamic traces. In: FSE (2008)
26. Gabel, M., Su, Z.: Symbolic mining of temporal specifications. In: ICSE (2008)
27. Godefroid, P.: Software model checking: The Verisoft approach. Formal Methods in System Design 26 (2005)
28. Henzinger, T.A., Jhala, R., Majumdar, R.: Permissive interfaces. In: ESEC/SIGSOFT FSE (2005)
29. Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Detecting malicious code by model checking. In: Julisch, K., Kruegel, C. (eds.) DIMVA 2005. LNCS, vol. 3548, pp. 174–187. Springer, Heidelberg (2005)
30. Kremenek, T., Twohey, P., Back, G., Ng, A.Y., Engler, D.R.: From uncertainty to belief: Inferring the specification within. In: OSDI (2006)
31. Kroening, D.: CBMC (2012), `http://www.cprover.org/cbmc`
32. Liu, C., Ye, E., Richardson, D.J.: Software library usage pattern extraction using a software model checker. In: ASE (2006)
33. Lo, D., Khoo, S.-C.: SMArTIC: towards building an accurate, robust and scalable specification miner. In: FSE 2006 (2006)
34. Lo, D., Ramalingam, G., Ranganath, V.P., Vaswani, K.: Mining quantified temporal rules: Formalism, algorithms, and evaluation. In: WCRE (2009)
35. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: ICSE 2008 (2008)
36. Ramanathan, M.K., Grama, A., Jagannathan, S.: Static specification inference using predicate mining. In: PLDI (2007)
37. Seshadri, P.: Generic Socket Programming tutorial (2008), `http://www.prasannatech.net/2008/07/socket-programming-tutorial.html`
38. Shoham, S., Yahav, E., Fink, S.J., Pistoia, M.: Static specification mining using automata-based abstractions. IEEE Trans. Software Eng. (2008)
39. Song, F., Touili, T.: Efficient CTL model-checking for pushdown systems. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 434–449. Springer, Heidelberg (2011)
40. Song, F., Touili, T.: Efficient malware detection using model-checking. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 418–433. Springer, Heidelberg (2012)
41. Song, F., Touili, T.: Pushdown model checking for malware detection. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 110–125. Springer, Heidelberg (2012)
42. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. (2003)
43. Visser, W., Mehlitz, P.C.: Model checking programs with java pathFinder. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 27–27. Springer, Heidelberg (2005)

44. Wasylkowski, A., Zeller, A.: Mining temporal specifications from object usage. Autom. Softw. Eng. (2011)
45. Wasylkowski, A., Zeller, A., Lindig, C.: Detecting object usage anomalies. In: ESEC/FSE (2007)
46. Witkowski, T., Blanc, N., Kroening, D., Weissenbacher, G.: Model checking concurrent linux device drivers. In: ASE (2007)
47. Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: mining temporal API rules from imperfect traces. In: ICSE (2006)

Formal Modelling and Verification of Population Protocols

Dominique Méry[1] and Michael Poppleton[2]

[1] LORIA & Université de Lorraine, BP 70239,
F-54506 Vandoeuvre lès Nancy, France
dominique.mery@loria.fr

[2] School of Electronics and Computer Science,
University of Southampton, Highfield,
Southampton SO17 1BJ, UK
mrp@ecs.soton.ac.uk

Abstract. The population protocols of Angluin, Aspnes et al [3] provide a theoretical framework for computability reasoning about algorithms for Mobile Ad-Hoc Networks (MANETs) and Wireless Sensor Networks (WSNs). By developing two example protocols and proving convergence results using the Event-B/RODIN [2] and TLA [11] frameworks, we explore the potential for formal analysis of these protocols.

1 Introduction

The design of a wireless sensor or mobile ad-hoc network (WSN/MANET) [7] for a given application requires demanding optimization against many parameters, e.g. node power and transmission range limits, variable node and link reliability, message latency and throughput [15]. A variety of energy-saving techniques and dynamic routing algorithms have been proposed. The verification of such long-running, time-dependent systems, with unreliable and dynamic substrate of computation/communication hardware, remains very challenging. While simulation [10] is the dominant design tool for these networks, formal methods are more recently beginning to be deployed.

A recent theoretical approach of interest is the population protocols of Angluin et al [3]. Assuming a finite population of agents interacting pairwise from some initial state, a class of protocols is defined that compute predicates over that state. Variants and extensions of this basic model, bringing it closer to real applications, have been proposed. The notion of global fairness - that interactions in these protocols are infinitely often enabled - is key to convergence arguments.

We discuss the formal development of two example population protocols. Experimental modelling, analysis and proof in the Event-B formal language and its RODIN toolkit [2] are straightforward. Event-B is a state-based formal specification language in first-order logic (FOL), supported by the rich RODIN toolkit of provers, animator, model checkers, graphical modelling front-ends, and infrastructural support for composition-decomposition in development. Functional and safety verification is provided by automatically generated proof obligations (PO) for invariant preservation and refinement.

The interesting questions about these protocols concern liveness and convergence properties and to what extent we can specify, reason about and prove such properties formally. A first-order scheme like Event-B cannot explicitly support this; we turn to Lamport's Temporal Logic of Actions (TLA). In the two example protocols we prove convergence properties using existing and new TLA fairness-based proof rules.

In the next section we give some background on population protocols, Event-B and TLA respectively. Sections 3 and 4 overview Event-B developments for two example protocols respectively; each shows a distinct style of reasoning about, and thus clarifying convergence properties. In the first refinement of section 3, the always-enabled character of the inductive step - two nodes interacting - means that convergence is proved inductively using the TLA weak fairness rule WF1. Further, the hypotheses of WF1 are first-order, thus expressible and provable in Event-B/RODIN. We have done one such proof, thus demonstrating an automated TLA-style proof of convergence. In the next refinement we introduce a dynamic interaction graph structure, so that the inductive step is no longer always enabled. This can be proved using the TLA strong fairness rule SF1; we show how this proof can be reduced to a WF1 proof by observing that the enablement of the inductive step is itself provable by WF1. The example suggests a first-order convergence proof method for a certain class of protocols; this remains to be investigated in future work.

Section 4 gives a more complex example and proposes a new notion of global fairness to prove convergence. Here, there are two stages of convergence; the first is proved using WF1 as before. We then overview two further proofs for the two alternate cases for the second stage; these proofs are given in detail in the companion paper [12]. The second stage proof requires the population protocol notion of *global* fairness [8]; we define - and prove sound in [12] - a new TLA proof rule GF1. Section 5 concludes.

2 Population Protocols, TLA and Event-B

2.1 The basic **population protocol** of Angluin et al [3] is a model of *passively mobile* simple devices with minimal storage. These agents are anonymous, and are passively mobile in the sense that any two agents may interact at any time. Choice/ scheduling of interacting pairs is outside the protocol. The interaction of any two agents is under a *global fairness* assumption [8], which expresses that an execution $E = C_0 \rightarrow C_1 \rightarrow \cdots$ is globally fair, when for every configuration C and C' such that $C \rightarrow C'$, if $C_i = C$ for infinitely many i, then $C_{i+1} = C'$ for infinitely many i.

Initially, each agent reads a single input and takes a corresponding initial state, after which no more input is read. The idea is for the protocol to compute a predicate over the multiset of input states. The system *stably converges* when all computations converge to a constant output vector. Note that the configuration of agent-states in such a computation need not converge, just the output.

Formally, a *population protocol* comprises finite input and output alphabets X and Y, a finite set Q of possible states of an agent, an input function $I : X \rightarrow Q$, output function $O : Q \rightarrow Y$, a transition function $\delta : Q \times Q \rightarrow Q \times Q$. A *population P* is a finite set V of agents with an irreflexive relation $E \subseteq V \times V$ which is the *interaction graph*. For [3] E is the complete interaction graph. A population *configuration* is a function $C : V \rightarrow Q$ giving the state of each node.

In this scheme, the protocol description is independent of population size, and thus storable on a small-memory device. Nodes have no identity, since that would increase with population size. Assuming that interacting pairs are scheduled randomly, independently and uniformly gives a *conjugating automaton*: this converges with probability 1, with expected number of interactions $O(n^2 \log n)$. [4] shows that it is *only* Pressburger[1] predicates that are computable in the basic population protocol.

[4] proposes various extensions of the basic model replacing immediate two-way interaction with one-way anonymous message-passing, immediate or delayed delivery, recording of sent messages, and queuing of incoming messages. In [5] the basic model is extended to describe *self-stabilizing* systems, where the protocol acts on input streams; the transition function becomes a relation $\delta : (Q \times X) \times (Q \times X) \to Q \times Q$.

2.2 Event-B is designed for long-running *reactive* hardware/software systems that respond to stimuli from user and/or environment. The set-theoretic language in first-order logic (FOL) takes as semantic model a transition system with guarded transitions between states. The correctness of a model is defined by an invariant property. The two units of structuring are the *machine* of dynamic variables, events and their invariants, and the *context* of static data of sets, constants and their axioms. Every machine *sees* at least one context.

The unit of behaviour is the *event*. An event acting on (a list of) state variables v, subject to enabling *guard* over local variable(s) t and state-updating *action*, has the following syntax and semantic model in a before-after predicate:

$$\text{ANY}\, t \text{ WHERE}\, Q(t,v) \text{ THEN}\, v := F(t,v) \text{ END} \quad \widehat{=}\ \exists t \cdot (Q(t,v) \wedge v' = F(t,v))$$

This defines a t-indexed nondeterministic choice between those transitions $v' = F(t,v)$ for which $Q(t,v)$ is true. t can be interpreted as either an input or an output to the event.

An event works in a model with constants c and sets s subject to *axioms* $P(s,c)$ and an *invariant* $I(s,c,v)$. Consistency proof obligations (POs) require that events are well-defined, feasible and maintain invariants. The term *refinement* is overloaded, referring both to the process of transforming models, and to the more concrete model which refines the abstract one. When model $N(w)$ refines $M(v)$, it has a refinement relation, or "gluing invariant" $J(s,c,v,w)$. New events may be introduced in refinement to act on new variables, effectively refining stuttering steps (called "skip" in Event-B). The refinement POs enforce the standard forward simulation refinement rule [1] that every concrete (refining) step of a refining or new event reestablishes the gluing invariant subject to some step of some abstract refined event, or skip.

A nondivergence PO requires that skip-refining (new) events do not take control forever: this is modelled using a VARIANT predicate. Every new event must must be proved to inductively reduce the variant, thus eventually disabling all such events.

2.3 Leslie Lamport's TLA (**Temporal Logic of Actions**) [11] is designed for the specification and verification of reactive systems in terms of their actions and behaviours (traces). It can be thought of as structured in four *tiers* [11]: (i) constants, and *constant formulas* - functions and predicates - over these, (ii) *state formulas* for reasoning about

[1] Pressburger arithmetic is a restricted integer arithmetic comprising 0, 1, $+$ and $<$.

states, expressed over *variables* as well as constants, (iii) *transition* or *action formulas* for reasoning about (before-after) pairs of states, and (iv) *temporal predicates* for reasoning about *behaviours*, i.e. traces of states; these are constructed from the other tiers and certain *temporal operators*.

An action formula expresses some fact or function about a system transition between one state and its successor, as made available by some system action. An action predicate is very like a before-after predicate in Event-B. A state formula is an action formula where either all flexible variables are unprimed, or all are primed. A state predicate is true in a behaviour iff it is true in the first state of that behaviour. If F, G are behaviour predicates then so are $\neg F, F \vee G, F \wedge G, F \Rightarrow G, \Box P, \Diamond P$. The latter two are temporal operators. We write $\Box P$ - called "always P" - to mean P is always true over a given behaviour, and define $\Diamond P$ - called "eventually P" - to be $\neg\Box\neg P$.

For action predicate A, list of state variables f, we define $[A]_f$ (called "square A sub f") to be true for states s, t iff $s[\![A \vee f' = f]\!]t$, that is, if either A defines a transition from s to t, or all variables f remain unchanged from s to t. Dually we define $\langle A \rangle_f$ (called "angle A sub f") to be true for states s, t iff $s[\![A \wedge f' \neq f]\!]t$, that is, A defines a transition from s to t, *and* at least one variable in f changes from s to t.

This logic enables us to specify the state-based temporal behaviour of a system, as well as assert properties over that behaviour, in one notation and logic. In general we wish to specify systems in the following form:

$$\Phi \mathrel{\hat{=}} Init_{\Phi} \wedge \Box[N]_f \wedge \mathrm{WF}_f(F_1) \wedge \mathrm{SF}_f(F_2)$$

$N \mathrel{\hat{=}} N_1 \vee N_2 \vee ...$ is the disjunction of all system actions, i.e. the "next" transition, denoting progress subject to possible stuttering. Stuttering is required to allow us to specify and prove refinements. $\Box Inv$ states an invariant *safety* property. The WF and SF constraints are the weak and strong fairness constraints required by the system actions in order to progress and F_1 and F_2 are two combinations of actions. We say that action A is weakly fair if, provided it is eventually always enabled, it is then guaranteed to fire infinitely often. A is strongly fair if, provided it is infinitely often enabled, it is then guaranteed to fire infinitely often. With the weaker antecedent in its implicative form, SF is the stronger fairness property. Global fairness GF is defined consistently with [8], and is a specialization of strong fairness, with explicit pre and postconditions P, Q' for A:

$$\begin{aligned}
\mathrm{WF}_f(A) &\mathrel{\hat{=}} \Diamond\Box Enabled\langle A \rangle_f \Rightarrow \Box\Diamond\langle A \rangle_f \\
\mathrm{SF}_f(A) &\mathrel{\hat{=}} \Box\Diamond Enabled\langle A \rangle_f \Rightarrow \Box\Diamond\langle A \rangle_f \\
\mathrm{GF}_f(P, A, Q) &\mathrel{\hat{=}} \Box\Diamond Enabled\langle P \wedge A \wedge Q' \rangle_f \Rightarrow \Box\Diamond\langle P \wedge A \wedge Q' \rangle_f
\end{aligned}$$

Consideration of whether an action eventually stabilises to always-enabled or not, determines the choice of a weak or strong fairness requirement in specification. This form of fairness specification places requirements on the scheduler of system actions. If the system environment is in scope its actions must be considered for fairness assumptions. Event-B - like other FOL model-based formalisms - does not express scheduling requirements. The variant mechanism in Event-B - to prevent non-divergence of new events in refinement - is relevant here. Such a variant eventually disables new events in favour of abstract, potentially blocked events. Of course, this simply enables, but does not guarantee, scheduling of a given abstract event.

Finally we define the *leads to* operator: $P \leadsto Q \;\widehat{=}\; \Box(P \Rightarrow \Diamond Q)$, meaning that whenever P holds then Q is guaranteed to hold at some later time.

Next we consider some of Lamport's proof rules for simple TLA [11]. LATTICE is an inductive proof rule for temporal reasoning: provided H_c leads to either the goal G or H_d for some d strictly smaller than c then the induction is guaranteed to converge to G.

WF1 gives the conditions under which weak fairness of action A is enough to guarantee that $P \leadsto Q$. A stuttering *progress step* produces either P or Q in the next state, nonstuttering action $\langle A\rangle_f$ takes the *inductive step* to produce Q, and under P, *inductive action* $\langle A\rangle_f$ is always enabled. SF1 is the strong fairness equivalent to prove $P \leadsto Q$: a strong fairness assumption on A is made and the same first two conditions hold as in WF1. An elaborated third condition ensures that $\langle A\rangle_f$ is *eventually* - rather than always - enabled. F may be required for expressing further fairness conditions.

LATTICE. $\succ$ a well-founded partial order on a set S

$$\frac{F \wedge c \in S \Rightarrow (H_c \leadsto (G \vee \exists d \in S \cdot (c \succ d) \wedge H_d))}{F \Rightarrow ((\exists c \in S \cdot H_c) \leadsto G)}$$

WF1.

$$\frac{\begin{array}{l} P \wedge [N]_f \Rightarrow (P' \vee Q') \\ P \wedge \langle N \wedge A\rangle_f \Rightarrow Q' \\ P \Rightarrow \mathit{Enabled}\langle A\rangle_f \end{array}}{\Box[N]_f \wedge \mathrm{WF}_f(A) \Rightarrow (P \leadsto Q)}$$

SF1.

$$\frac{\begin{array}{l} P \wedge [N]_f \Rightarrow (P' \vee Q') \\ P \wedge \langle N \wedge A\rangle_f \Rightarrow Q' \\ \Box P \wedge \Box[N]_f \wedge \Box F \Rightarrow \Diamond \mathit{Enabled}\langle A\rangle_f \end{array}}{\Box[N]_f \wedge \mathrm{SF}_f(A) \wedge \Box F \Rightarrow (P \leadsto Q)}$$

3 Red and Green Lights

We present a simple population protocol model in Event-B and some refinements, in order to demonstrate a temporal style of reasoning about convergence. Nodes ll are coloured red or green (coded $ll \in V \rightarrow \mathsf{COLOURS}$ where $\mathsf{COLOURS} = \{green, red\}$). In an interaction, if any two red nodes are adjacent - i.e. connected by the graph - then one will turn green. The protocol terminates when only one red remains.

In the abstract model M0, the graph is complete - every node is connected to every other. Nodes are initialised arbitrarily. In one shot, event *conv* nondeterministically assigns one node to red and all others to green.

```
M0: EVENT conv
      ANY    $i$
      WHERE  $i \in V$
      THEN   $ll := ((V \setminus \{i\}) \times \{green\}) \cup \{i \mapsto red\}$
      END
```

In refinement M1, the new event iact pairwise switches one red node of an adjacent red pair to green, subject to an obvious variant (the *convergent* keyword generates an inductive PO for iact). conv skips, marking convergence to a single red node. Event-B refinement allows strengthening of guards, as long as the overall system guard is maintained; there are associated proof obligations.

```
M1: EVENT conv
      REFINES   $conv$
      ANY   $i$
      WHERE
        $i \in V \wedge ll(i) = red$
        $\wedge\ ran(i \lhd ll) = \{green\}$
      THEN   $skip$
      END
```

```
M1: EVENT iact   $convergent$
      ANY  $i\ j$
      WHERE
        $i \in V \wedge j \in V \wedge i \neq j \wedge ll(i) = red \wedge ll(j) = red$
      THEN  $ll(i) := green$
      END

      VARIANT  $ll \rhd \{red\}$
```

Convergence is proved using Lamport's fairness proof rules - rather trivially, for only one event apart from initialisation and termination. In M1, it is obvious that each interaction iact reduces the problem and that $ll \rhd \{red\}$ is a suitable set-valued inductive/variant expression. Using TLA we can be more explicit about the inductive process of convergence than we can in Event-B - define:

$$P \mathrel{\widehat{=}} Inv_{M1} \wedge R(n+1) \qquad Q \mathrel{\widehat{=}} Inv_{M1} \wedge R(n)$$

$$Inv_{M1} \mathrel{\widehat{=}} ll \in V \to \mathsf{COLOURS} \qquad R(n+1) \mathrel{\widehat{=}} card(ll^{-1}[\{red\}] = n+1$$

$$N \mathrel{\widehat{=}} iact \vee conv \qquad A_{iact} \mathrel{\widehat{=}} iact$$

Applying WF1 ...

$$\begin{array}{ll} P \wedge [N]_{ll} \Rightarrow (P' \vee Q') & \text{progress} \\ P \wedge \langle N \wedge A_{iact}\rangle_{ll} \Rightarrow Q' & \text{inductive step} \\ P \Rightarrow Enabled\langle A_{iact}\rangle_{ll} & \text{inductive action} \\ \hline \Box[N]_{ll} \wedge \mathrm{WF}_{ll}(A_{iact}) \Rightarrow (P \leadsto Q) & \end{array}$$

Inductive action A_{iact} establishes $Q' = Inv'_{M1} \wedge R'(n)$. Thus assuming weak fairness of $iact$ we can prove $R(n+1) \leadsto R(n)$, and apply induction by LATTICE to prove convergence to $R(1)$. Weak fairness suffices for interaction to happen infinitely often since iact is always enabled for $n \geq 1$. Note, in this simple example, that the three hypotheses of WF1 contain no temporal operators and are thus all statements of first-order logic. They are therefore expressible and provable in Event-B/RODIN; this we have done for this M1 proof.

In refinements M2 and M3, we add a new variable $conn \in V \leftrightarrow V$ to model the dynamically connected graph, initialised arbitrarily. The iact guard is refined allowing only connected nodes to interact. A new event daemon in M2 arbitrarily adds links to, or removes links from the graph. In M3 the daemon is refined into an angel, which connects two red nodes in the graph where it can, and a daemon, which arbitrarily removes links, or connects two nodes when at least one is green.

```
M3: EVENT angel REFINES daemon
      ANY  $i\ j$
      WHERE  $i \in V \wedge j \in V \wedge i \neq j \wedge ll(i) = red \wedge ll(j) = red$
      THEN
        $conn := conn \cup \{i \mapsto j\}$
      END

M3: EVENT daemon REFINES daemon
      ANY  $i\ j$
      WHERE  $i \in V \wedge j \in V \wedge i \neq j$
      THEN
        $conn :| (i \mapsto j \notin conn \wedge (ll(i) = green \vee ll(j) = green)$
                        $\wedge\, conn' = conn \cup \{i \mapsto j\})$
                  $\vee (i \mapsto j \in conn \wedge conn' = conn \setminus \{i \mapsto j\})$
      END
```

Consider the M3 scenario. Our simple angel-daemon model of the environment's dynamic disruption of the network is essentially nondeterministic; an implicit variant such as in M1 is not available. The variant-based convergence proof required by Event-B effectively forces us to schedule the environment here explicitly, perhaps designing in some counter or time bound on which to base a variant. This is too concrete a view of scheduling, and TLA allows more flexible and abstract reasoning about scheduling and convergence. Note that iact is no longer always enabled since two reds may not be connected at a given time. It may be infinitely often disabled, thus we need strong fairness for iact. Subject to the following definitions, apply SF1:

$$\begin{array}{ll}
P \mathrel{\widehat{=}} Inv_{M3} \wedge R(n+1) & Q_{iact} \mathrel{\widehat{=}} Inv_{M3} \wedge R(n) \\
Inv_{M3} \mathrel{\widehat{=}} \begin{pmatrix} ll \in V \to \mathsf{COLOURS} \\ \wedge\, conn \in V \leftrightarrow V \\ \wedge\, conn \cap id = \varnothing \end{pmatrix} & R(n+1) \mathrel{\widehat{=}} card(ll^{-1}[\{red\}] = n+1 \\
N \mathrel{\widehat{=}} iact \vee angel \vee daemon \vee conv & A_{iact} \mathrel{\widehat{=}} iact
\end{array}$$

$$\begin{array}{ll}
 & \text{Applying SF1 ...} \\
\text{SF1.1} & P \wedge [N]_{ll} \Rightarrow (P' \vee Q'_{iact}) \\
\text{SF1.2} & P \wedge \langle N \wedge A_{iact} \rangle_{ll} \Rightarrow Q'_{iact} \\
\text{SF1.3} & \Box P \wedge \Box[N]_{ll} \Rightarrow \Diamond Enabled\langle A_{iact} \rangle_{ll} \\
\hline
\text{SF1.C} & \Box[N]_{ll} \wedge \text{SF}_f(A_{iact}) \Rightarrow (P \leadsto Q_{iact})
\end{array}$$

SF1.1 and SF1.2 are similar to the WF proof above, differing only in the stronger invariant of M3. SF1.3 is a formula in temporal logic not directly expressible in Event-B. However we observe that this hypothesis can be expressed in the stronger *leadsto* form:

$$\begin{array}{lll}
 & \Box[N]_{ll} \Rightarrow (\Box P \Rightarrow \Diamond Enabled\langle A_{iact} \rangle_{ll}) & \\
\Leftarrow & \Box[N]_{ll} \Rightarrow P \leadsto Enabled\langle A_{iact} \rangle_{ll} & \mathrel{\widehat{=}} \text{SF1.3'}
\end{array}$$

That is, if SF1.3' - that $\langle A_{iact} \rangle_{ll}$ is eventually enabled by some event - can be proved in first-order logic by some fairness argument like that in the previous section, we are done. In M3 the environment's dynamic effect on the network is modelled by the daemon and - helped by the support team - the angel. We regard the angel as always enabled, thus a weak fairness assumption suffices to ensure it acts infinitely often. We adjust our above definitions now to prove SF1.3' by the weak fairness of the angel - the trigger event is now the angel $A_{angel} = angel$ and the target state is Q_{angel}, the enablement of interaction event $iact$:

$$Q_{angel} \mathrel{\widehat{=}} Enabled\langle A_{iact}\rangle_{ll} \qquad A_{angel} \mathrel{\widehat{=}} angel$$

... giving

$$\begin{array}{ll} & P \wedge [N]_{ll} \Rightarrow (P' \vee Q'_{angel}) \\ & P \wedge \langle N \wedge A_{angel}\rangle_{ll} \Rightarrow Q'_{angel} \\ & P \Rightarrow Enabled\langle A_{angel}\rangle_{ll} \\ \hline \text{SF1.3'} & \Box[N]_{ll} \wedge \text{WF}_{ll}(A_{angel}) \Rightarrow (P \leadsto Q_{angel}) \end{array}$$

It is useful finally to add one more step in the direction of realism in this example. Whereas the daemon of environmental conditions or damage may reasonably be assumed to be always enabled, the angel may not: bad weather conditions for node - node radio transmission take time to clear, as does a maintenance team to replace batteries on nodes. It is thus more realistic to place a strong fairness requirement on a sometimes-enabled angel. We then find that the analogous proof of SF1.3' of the above becomes a strong fairness proof - thus generating another, secondary proof obligation SF1.3" on the enablement of the angel:

$$\Box[N]_{ll} \wedge \Box F \Rightarrow P \leadsto Enabled\langle A_{angel}\rangle_{ll}$$

This process suggests a recursive first-order proof method - provided the recursion terminates with some initial, weakly fair triggering action.

4 The Dancers

A group of dancers [6] are each marked as either follower(F) or leader(L). The aim of this protocol is to reach a configuration where if there are (i) initially more leaders than followers, then $\#(leaders - followers)$ leaders and no followers remain, (ii) initially more followers than leaders, then $\#(followers - leaders)$ followers and no leaders remain, (iii) initially equal numbers of followers and leaders, then none of either remain. The target configuration is reached by applying the following transitions:

$$F \leftrightarrow L \Rightarrow 0 \leftrightarrow 0 \quad L \leftrightarrow 0 \Rightarrow L \leftrightarrow 1 \quad 0 \leftrightarrow 1 \Rightarrow 0 \leftrightarrow 0 \quad F \leftrightarrow 1 \Rightarrow F \leftrightarrow 0$$

We show that this protocol *eventually* leads to a configuration in which there are only X dancers, O dancers and U dancers where X is either a set of followers or a set of leaders, U is a set of one dancers, and O is a set of $zero$ dancers.

The set D of all dancers is initially partitioned into F_0 the initial set of followers, L_0 the initial set of leaders, O_0 the initial set of $zero$ dancers and U_0 the initial set of one dancers. In the original problem [6], the sets U_0 and O_0 are empty but here we generalize the problem. In temporal language the first property to verify is :

$$partition(D, L_0, F_0, U_0, O_0) \leadsto \exists X, U, O. \begin{pmatrix} partition(D, X, U, O) \\ \wedge (X \subseteq F_0 \vee X \subseteq L_0) \\ \wedge O_0 \cup U_0 \subseteq O \cup U \end{pmatrix} \qquad (1)$$

Our first model PopDancers (PD(1)) starts by defining abstract events which in one shot nondeterministically assign to the appropriate case: either no leader or no follower. Event Followers applies when there are as many, or fewer leaders than followers: there is an injection i from L_0 into F_0. Event Leaders applies when the number of leaders is strictly greater than the number of followers: there is an injection i from F_0 into L_0.

```
PD(1) : EVENT Followers
  ANY  $i$
  WHERE
    $F = F0 \wedge L = L0$
    $i \in L0 \rightarrowtail F0$
  THEN
    $U, O, L, F :|$
    $\left(\begin{array}{l}(partition(D, F', L', O', U') \\ \wedge\ L' = \varnothing \wedge F' = F \setminus i[L] \\ \wedge\ O0 \cup U0 \subseteq O' \cup U'))\end{array}\right)$
  END
```

```
PD(1) : EVENT Leaders
  ANY  $i$
  WHERE
    $L = L0 \wedge F = F0$
    $i \in F0 \rightarrowtail L0 \wedge i[F0] \neq L0$
  THEN
    $U, O, L, F :|$
    $\left(\begin{array}{l}(partition(D, F', L', O', U') \\ \wedge\ F' = \varnothing \wedge L' = L \setminus i[F] \\ \wedge\ O0 \cup U0 \subseteq O' \cup U'))\end{array}\right)$
  END
```

$$partition(D, F, L, O, U) \wedge F \subseteq F0 \wedge L \subseteq L0 \wedge O0 \cup U0 \subseteq O \cup U \qquad (2)$$

This initial model asserts the existence of an injection from one set of dancers into the other. The algorithmic process will progressively construct the final injection. By definition model PopDancers with invariant (2) satisfies property (1).

4.1 The Algorithmic Process

Model PopDancers is refined by PopDancing (PD(2)), which introduces the algorithm for getting a configuration satisfying (1). We introduce new variables vf, vl, vu, vo, f, l as follows. vl, vf initially contain L_0, F_0 respectively; L, F are respectively assigned to the final values of these variables. vu, vo contain one-dancers and zero-dancers respectively. f, l each record the injection required for the refinement; these functions are constructed iteratively and only one is finally used, since either followers or leaders win.

Considering the transition rules, we see that when a dancer of vl or vf moves into vo, he/she will never return to vl or vf. We can derive an inductive property based on $vl \cup vf$ - with followers and leaders remaining - expressing the fact that the set $vl \cup vf$ is strictly decreasing by the rule $F \leftrightarrow L \Rightarrow 0 \leftrightarrow 0$. We express this as an intermediate liveness property (3):

$$\mathsf{DANCING}(vl, vf, vu, vo) \overset{def}{=} partition(D, vl, vf, vu, vo)$$

$$\left(\begin{array}{l}\mathsf{DANCING}(vl, vf, vu, vo) \\ \wedge\ vl \neq \varnothing \wedge vf \neq \varnothing\end{array}\right) \leadsto \exists \left(\begin{array}{l}vl', vf' \\ vu', vo'\end{array}\right) \cdot \left(\begin{array}{l}\mathsf{DANCING}(vl', vf', vu', vo') \\ \wedge vl' \cup vf' \subset vl \cup vf\end{array}\right) \qquad (3)$$

Property (3) is proved by applying the WF1 rule of TLA as follows. The inductive step (4) is given by event Dancing which models the transformation of an L-F pair in $vl \cup vf$ into a 0-0 pair in vo. Dancing should have a weak fairness assumption to ensure progress. The progress step (5) ensures that no event increases the dancers of $vl \cup vf$. NB: notation $BA(\mathrm{e})(h, h')$ denotes the before-after relation of event e in the frame h.

$$\begin{pmatrix} \mathsf{DANCING}(vl, vf, vu, vo) \\ \wedge vl \neq \varnothing \\ \wedge vf \neq \varnothing \\ \wedge h = (vl, vf, vu, vo) \\ \wedge BA(\mathsf{Dancing})(h, h') \end{pmatrix} \Rightarrow \begin{array}{l} \mathsf{DANCING}(vl', vf', vu', vo') \\ \wedge\, vl' \cup vf' \subset vl \cup vf \end{array} \tag{4}$$

$$\begin{array}{l} \forall e.e \in \{\mathsf{Dancing}, \mathsf{DancingOU}, \\ \quad \mathsf{DancingFU}, \mathsf{DancingLO}\} \wedge \\ \begin{pmatrix} \mathsf{DANCING}(vl, vf, vu, vo) \\ \wedge vl \neq \varnothing \\ \wedge vf \neq \varnothing \\ \wedge h = (vl, vf, vu, vo) \\ \wedge card(L_0) < card(F_0) \\ \wedge BA(e)(h, h') \end{pmatrix} \\ \Rightarrow \begin{array}{l} \mathsf{DANCING}(vl', vf', vu', vo') \\ \wedge vl' \cup vf' \subseteq vl \cup vf \end{array} \end{array} \tag{5}$$

Given these two conditions, and under the assumption of (3) event Dancing is always enabled, we can infer liveness property (3). Using the LATTICE induction rule $vl \cup vf$ will strictly reduce at each Dancing step until either it becomes $\varnothing$ or an hypothesis of (3) goes false, i.e. when either $vl = \varnothing$ or $vf = \varnothing$. Thus the convergence property (1) is proved, QED.

Next we give the events of this model PopDancing.

$inv1 : l \in F0 \nrightarrow L0 \wedge f \in L0 \nrightarrow F0$
$inv2 : f \neq \varnothing \Rightarrow f \in dom(f) \twoheadrightarrow\!\!\!\!\!\!\!\!\nrightarrow ran(f) \wedge f \in L0 \rightarrowtail\!\!\!\!\!\nrightarrow F0$
$inv5 : l \in F0 \rightarrowtail\!\!\!\!\!\nrightarrow L0 \wedge l \neq \varnothing \Rightarrow l \in dom(l) \twoheadrightarrow\!\!\!\!\!\!\!\!\nrightarrow ran(l)$
$inv21 : partition(D, vf, vl, vo, vu)$
$inv11 : vl \subseteq L0 \wedge vf \subseteq F0 \wedge inv14 : oldf \subseteq F0 \wedge inv15 : oldl \subseteq L0 \wedge vl \cup oldl = L0$
$inv17 : vf \cup oldf = F0 \wedge dom(l) = oldf \wedge ran(l) = oldl \wedge l \in oldf \rightarrowtail oldl \wedge l \in oldf \twoheadrightarrow oldl$
$inv19 : dom(f) = oldl \wedge ran(f) = oldf \wedge f \in oldl \rightarrowtail oldf \wedge f \in oldl \twoheadrightarrow oldf$
$inv22 : vf \cap oldf = \varnothing \wedge vl \cap oldl = \varnothing \wedge U0 \subseteq vo \cup vu \wedge O0 \subseteq vo \cup vu \wedge vf \cup oldf = F0$
$inv31 : vl \cup oldl = L0 \wedge vf = F0 \setminus oldf \wedge vl = L0 \setminus oldl \wedge oldl = l[oldf] \wedge : oldf = f[oldl]$
$inv38 : f = l^{-1} \wedge l = f^{-1}$
$inv40 : vl = \varnothing \wedge f \neq \varnothing \Rightarrow f \in L0 \rightarrowtail F0$
$inv41 : vf = \varnothing \wedge l \neq \varnothing \Rightarrow l \in F0 \rightarrowtail L0$
$inv42 : end \in BOOL$
$inv43 : end = FALSE \Rightarrow F = F0 \wedge L = L0$
$inv44 : end = TRUE \wedge L \neq \varnothing \Rightarrow F = \varnothing$
$inv45 : end = TRUE \wedge F \neq \varnothing \Rightarrow L = \varnothing$

Fig. 1. Invariant for the model PopDancing

```
PD(2) : EVENT Dancing
  ANY
    $x, y$
  WHERE
    $grd1 : x \in vf \wedge y \in vl$
  THEN
    $act1 : vo := vo \cup \{x, y\}$
    $act2 : vf := vf \setminus \{x\}$
    $act3 : vl := vl \setminus \{y\}$
    $act4 : oldf := oldf \cup \{x\}$
    $act5 : oldl := oldl \cup \{y\}$
    $act6 : f(y) := x$
    $act7 : l(x) := y$
  END
```

Event Dancing, guarded on the existence of both followers and leaders, models the transition rule $F \leftrightarrow L \Rightarrow 0 \leftrightarrow 0$. It modifies vo, vf, vl while building both the injections f and l. The three next events model the three transition rules $F \leftrightarrow 1 \Rightarrow F \leftrightarrow 0,\ L \leftrightarrow 0 \Rightarrow L \leftrightarrow 1$ and $0 \leftrightarrow 1 \Rightarrow 0 \leftrightarrow 0$ respectively over vo and vu.

```
PD(2) : EVENT DancingFU
  ANY
    $x, y$
  WHERE
    $grd1 : x \in vf \wedge y \in vu$
  THEN
    $act1 : vo := vo \cup \{y\}$
    $act2 : vu := vu \setminus \{y\}$
  END
```

```
PD(2) : EVENT DancingLO
  ANY
    $x, y$
  WHERE
    $grd1 : x \in vl \wedge y \in vo$
  THEN
    $act1 : vo := vo \setminus \{y\}$
    $act2 : vu := vu \cup \{y\}$
  END
```

```
PD(2) : EVENT DancingOU
  ANY
    $x, y$
  WHERE
    $grd1 : x \in vo \wedge y \in vu$
  THEN
    $act1 : vo := vo \cup \{y\}$
    $act2 : vu := vu \setminus \{y\}$
  END
```

This is still an abstract model; not yet sufficiently refined to merge termination in one unique concrete event. The two events Followers and Leaders are modelling the end of the construction of the injection: either f or l. The Event-B refinement witness mechanism is used to implement the abstract nondeterministic choice of injection i in PopDancers/Leaders by the concrete injection value l in PopDancing/Leaders below.

The next step is to refine the current model into the concrete version of a *codable* Event-B model, equivalent to the set of transition rules that define the population protocol.

```
PD(2) : EVENT Leaders
  REFINES Leaders
  WHEN
    $grd3 : vf = \varnothing \wedge vl \neq \varnothing$
    $grd4 : end = FALSE$
  WITNESSES
    $i : i = l$
  THEN
    $act4 : U := vu$
    $act5 : O := vo$
    $act6 : F := vf$
    $act7 : L := vl$
    $act8 : end := TRUE$
  END
```

```
PD(2) : EVENT Followers
  REFINES Followers
  WHEN
    $grd4 : vl = \varnothing$
    $grd5 : end = FALSE$
  WITNESSES
    $i : i = f$
  THEN
    $act4 : U := vu$
    $act5 : O := vo$
    $act6 : F := vf$
    $act7 : L := vl$
    $act8 : end := TRUE$
  END
```

4.2 Generating the Population Protocol from Refinement

The last model is called Protocol (PD(3)) and is described as follows:

```
PD(3) : EVENT Termination
  REFINES Leaders Followers
  WHEN
    $grd3 : vf = \varnothing \vee vl = \varnothing$
    $grd5 : end = FALSE$
  THEN
    $act4 : U := vu$
    $act5 : O := vo$
    $act6 : F := vf$
    $act7 : L := vl$
    $act8 : end := TRUE$
  END
```

```
PD(3) : EVENT Dancing
  REFINES Dancing
  ANY
    $x, y$
  WHERE
    $grd1 : x \in vf$
    $grd2 : y \in vl$
  THEN
    $act1 : vo := vo \cup \{x, y\}$
    $act2 : vf := vf \setminus \{x\}$
    $act3 : vl := vl \setminus \{y\}$
  END
```

Event Termination models the global termination of the process; it is not an action of the protocol itself but only an observation by a global observer. The condition end is set to true at this step. It refines two events by merging them. Event Dancing is transformed into a simplified rule $F \leftrightarrow L \Rightarrow 0 \leftrightarrow 0$, and the other rule events DancingFU, DancingLO and DancingOU remain unchanged

4.3 Temporal Analysis of Fairness Requirements

The Event-B model of the previous section expresses the population protocol rules, and does not express any assumptions over executions or scheduling. Indeed, such statements are not possible in Event-B, which is a language of single-step state transitions.

We now analyse fairness conditions to prove that the protocol reaches a stable configuration. Possible configurations D can be described as follows:

- $F \oplus L \oplus O \oplus U$: at least one each of Follower, Leader, ZERO and ONE.
- $F \oplus O \oplus U$: at least one each of Follower, ZERO and ONE, and no Leader.
- $L \oplus O \oplus U$: at least one each of Leader, ZERO and ONE, and no Follower.
- $L \oplus U$: at least one each of Leader and ONE, and no Follower and no ZERO.
- $F \oplus O$: at least one each of Follower and ZERO, and no Leader and no ONE.
- ... $\oplus\, T \oplus$ ... where T = 0 or 1 means that there is exactly one ZERO or ONE.
- $L \oplus 0^i \oplus 1^j$: at least one Leader, and i ZEROs and j ONEs.

We can regard these configurations as liveness properties, and use a predicate diagram [9] to describe progress through them. Each arrow states a $\leadsto$ property. The first liveness property is in fact (1): $D = F \oplus L \oplus O \oplus U \leadsto \exists X.D = X \oplus O \oplus U$ which is split into two cases corresponding to X as F or L. The previous section proved this property, showing that assuming event Dancing is weakly fair, the appropriate case is eventually reached.

$$\begin{tikzcd}
 & D = F \oplus L \oplus O \oplus U \arrow[dl] \arrow[dr] & \\
D = F \oplus O \oplus U \arrow[d] & & D = L \oplus O \oplus U \arrow[d] \\
D = F \oplus O & & D = L \oplus U
\end{tikzcd}$$

We describe the two target configurations $D = F \oplus O$ and $D = L \oplus U$ as *stable* in the sense of not changing further, and prove that they are reachable using fairness assumptions. We consider each case separately.

Case 1: $D = F \oplus O \oplus U \leadsto D = F \oplus O$. A full proof is given in the companion paper [12]. It is a straightforward WF1 argument similar to that of section 4.1, using a weak fairness assumption on DancingFU, DancingOU, each of which reduce vu. These are the only transitions enabled in this case.

Case 2: $D = L \oplus O \oplus U \leadsto D = L \oplus U$. A full proof is given in the companion paper [12]. As before there are only two enabled events in this case: DancingLO, DancingOU. However this proof is a more complex argument than case 1 and requires richer fairness assumptions because of the way DancingOU consumes the ONEs produced by DancingLO. We see looping transitions through the intermediate configurations and note that the configuration $D = L \oplus U$ is stable once reached, because

DancingOU becomes disabled by the absence of ZEROs:

$$L \oplus 0^i \oplus 1^j \overset{DancingLO}{\longrightarrow} L \oplus 0^{i-1} \oplus 1^{j+1} \overset{DancingLO}{\longrightarrow} \cdots L \oplus 1^{i+j}$$
$$L \oplus 0^i \oplus 1^j \overset{DancingOU}{\longleftarrow} L \oplus 0^{i-1} \oplus 1^{j+1}$$

Global fairness states that, if a configuration C appears infinitely often in a sequence of configurations, and if $C \longrightarrow C'$, then C' should appear also infinitely often in the sequence. This is expressed using our global fairness rule GFd_h applied to the dancers:

$$\begin{array}{ll}\mathsf{GFd}_h \overset{def}{=} & \forall i, j \cdot i, j \in 1..n \wedge i + j = card(O \cup U) = n \\ & \Rightarrow \mathrm{GF}_h(D = L \oplus 0^i \oplus 1^j, \mathsf{DancingLO}, D = L \oplus 0^{i-1} \oplus 1^{j+1})\end{array}$$

The question is to integrate the global fairness assumption in the reasoning. The problem to solve can be summarized by the following diagram which gives the possible transitions among the possible configurations:

$0^n \leftrightarrows 0^{n-1}1^1 \leftrightarrows 0^{n-2}1^2 \leftrightarrows \ldots \leftrightarrows 0^2 1^{n-2} \leftrightarrows 01^{n-1} \rightarrow 1^n$

The target configuration is the configuration in which there is no more O element. The global fairness assumption means that each triple of configuration of the sequence above is infinitely often enabled, since the number of triples is finite. Intuitively, under the global fairness, the target configuration is reached.

We propose a new rule GF1 for deriving liveness properties under the global fairness assumption. The rule is based on the WF1 and SF1 rules of Lamport[11]. It extends WF1 with another configuration: from B we may progress to B', take the inductive step E to C', or reach another configuration A' which works against the inductive process. This "counter-inductive" step is itself counteracted by an assumption that, given GF_f, $A \leadsto B$. E should be enabled in B.

$$\begin{array}{ll} \mathrm{GF1} & B \wedge [N]_f \Rightarrow (B' \vee C' \vee A') \\ & B \wedge \langle N \wedge E \rangle_f \Rightarrow C' \\ & \Box[N]_f \wedge \mathrm{GF}_f(A, E, B) \Rightarrow (A \leadsto B) \\ & B \Rightarrow Enabled\langle E \rangle_f \\ \hline & \Box[N]_f \wedge \mathrm{GF}_f(A, E, B) \Rightarrow (B \leadsto C) \end{array}$$

The global fairness assumption is defined over (configuration, event, configuration) tuples, unlike the classical fairness assumptions made on *actions* or *events* in TLA. The classical WF/SF proof rules of TLA are not enough to prove convergence to the configuration $D = L \oplus U$.

5 Conclusion

We have performed Event-B developments for two example population protocols and fully discharged the usual first-order proof obligations in the RODIN toolkit. We cannot however directly prove - or even specify, for that matter - liveness and convergence properties for these protocols in this first-order formal language. We have shown how the standard WF/SF proof rules of TLA can be applied to such liveness and convergence proofs. In the first protocol we found a style of reasoning where by treating enablement of the inductive step as an intermediate liveness property, we could express these rules in FOL. We saw that such a proof could thus be automated using the RODIN provers. The second protocol gave further examples of the WF style of proof, and demonstrated the use of global fairness in a more intricate convergence proof.

Both examples illustrate how Event-B can be integrated into a general framework dealing with trace semantics and fairness assumptions. New proof obligations are reduced to FOL proofs upto temporal reasoning using a limited set of temporal proof rules borrowed to TLA. In this way, we obtain *for free* a TLA-based trace semantics, which can be managed by the RODIN toolkit with minor modifications. The integration is driven by the design of wireless sensor or mobile ad-hoc networks (WSN/MANET) [7] and it provides a framework integrating the refinement of Event-B and the expressivity of TLA.

Immediate tasks are to characterise the class of problems tractable to such reasoning, and to establish the cases when the proofs can be made first-order and thus automatable in tools like RODIN.

Having demonstrated the utility of Event-B modelling with TLA reasoning for simple algorithms, further work could tackle (i) the extended population protocol models of section 2 and even more challenging (ii) a real WSN/MANET algorithm. Two interesting application candidates are data aggregation [13] and localisation [14]. Aggregation is concerned with reducing data traffic either by averaging sensor data on a regional basis, or simply packing readings into larger messages. This includes notions of routing from data source to sink. In localisation, each node must dynamically identify neighbours to whom it is connected.

References

[1] Abrial, J.-R.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010)

[2] Abrial, J.-R., Butler, M., Hallerstede, S., Voisin, L.: An open extensible tool environment for event-B. In: Liu, Z., Kleinberg, R.D. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 588–605. Springer, Heidelberg (2006)

[3] Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. Distributed Computing 18(4), 235–253 (2006)

[4] Angluin, D., Aspnes, J., Eisenstat, D., Ruppert, E.: The computational power of population protocols. Distributed Computing 20(4), 279–304 (2007)

[5] Angluin, D., Aspnes, J., Fischer, M.J., Jiang, H.: Self-stabilizing population protocols. TAAS 3(4) (2008)

[6] Aspnes, J., Ruppert, E.: An introduction to population protocols. Bulletin of the EATCS 93, 98–117 (2007)

[7] Banatre, M., Ollero, A., Wolisz, A.: Cooperating Embedded Systems and Wireless Sensor Networks. John Wiley (2008)

[8] Cai, S., Izumi, T., Wada, K.: How to prove impossibility under global fairness: On space complexity of self-stabilizing leader election on a population protocol model. Theory Comput. Syst. 50(3), 433–445 (2012)

[9] Cansell, D., Méry, D., Merz, S.: Predicate diagrams for the verification of reactive systems. In: Grieskamp, W., Santen, T., Stoddart, B. (eds.) IFM 2000. LNCS, vol. 1945, pp. 380–397. Springer, Heidelberg (2000)

[10] Egea-López, E., Vales-Alonso, J., Martínez-Sala, A.S., Pavón-Mariño, P., García Haro, J.: Simulation tools for wireless sensor networks. In: SPECTS 2005: Summer Simulation Multiconference (2005)

[11] Lamport, L.: The Temporal Logic of Actions. ACM Trans. Program. Lang. Syst. 16(3), 872–923 (1994)

[12] Méry, D., Poppleton, M.: Formal modelling and verification of population protocols. Technical report, LORIA (2013)

[13] Rajagopalan, R., Varshney, P.K.: Data-aggregation techniques in sensor networks: A survey. IEEE Communications Surveys and Tutorials 8(1-4), 48–63 (2006)

[14] Stavvides, A., Srivastava, M., Girod, L., Estrin, D.: Wireless Sensor Networks. Springer (2004)

[15] Woo, A., Tong, T., Culler, D.: Taming the underlying challenges of reliable multihop routing in sensor networks. In: Proceedings of the 1st International Conference on Embedded Networked Sensor Systems, SenSys 2003, pp. 14–27. ACM, New York (2003)

Detecting Vulnerabilities in Java-Card Bytecode Verifiers Using Model-Based Testing

Aymerick Savary[1,2], Marc Frappier[1], and Jean-Louis Lanet[2]

[1] University of Sherbrooke
[2] University of Limoges

Abstract. Java Card security is based on different elements among which the bytecode verifier is one of the most important. Finding vulnerabilities is a complex, tedious and error-prone task. In the case of the Java bytecode verifier, vulnerability tests are typically derived by hand. We propose a new approach to generate vulnerability test suites using model-based testing. Each instruction of the Java bytecode language is represented by an event of an Event-B machine, with a guard that denotes security conditions as defined in the virtual machine specification. We generate vulnerability tests by negating guards of events and generating traces with these faulty events using the ProB model checker. This approach has been applied to a subset of twelve instructions of the bytecode language and tested on five Java Card bytecode verifiers. Vulnerabilities have been found for each of them. We have developed a complete tool chain to support the approach and provide a proof of concept.

Keywords: Model Based Testing, Java Card bytecode Verifier, Vulnerability Testing, Security, Event-B.

1 Introduction

The Java Card technology [18] is a subset of the Java platform [16] that enables Java programs to run on resource constrained platforms like smart cards and other small devices. Security is an important concern in this platform, and it is ensured through various mechanisms *i.e.,* the firewall, the bytecode Verifier (BCV), the sharing mechanism... The firewall is in charge to provide segregation between different application providers. The sharing mechanism implements a security policy using an identity based protocol to allow information flows between different application providers. The BCV checks by static analysis that a Java Card program satisfies security constraints defined in the Java Virtual Machine specification (JVM). In the last version, 3.0.4 *Connected Edition* of the Java Card technology, the BCV is mandatory and must be executed on the smart card. This raises the issue of checking the correctness of the embedded BCV. Since a smart card has limited resources, developers may be tempted to optimize the BCV, possibly introducing subtle errors through complex optimization techniques. Testing such devices is a delicate and time consuming task.

Thus, special care must be taken to ensure good coverage while minimizing the number of test cases, because testing such embedded systems is more laborious than stand-alone software systems.

BCV test cases are typically derived by hand. In this paper, we propose an approach to automate the generation of BCV test cases. We distinguish two classes of tests: i) conformance tests: they ensure that correct bytecode programs are indeed accepted by the BCV; ii) vulnerability tests: they ensure that incorrect Java bytecode programs are indeed rejected by the BCV. Detecting vulnerabilities is critical from a security point of view, because accepting incorrect programs may lead to attacks. These two classes require different test generation strategies; in this paper, we focus on the generation of vulnerability tests.

We adopt a model-based approach for the generation of vulnerability tests. This is an alternative to the proof and code generation method used by [5]. The idea is to model each function of the program (in this case each instruction of the language) by an event of an Event-B model. The event's guard represents the precondition of the instruction as defined in the JVM specification, which expresses the security constraints on an instruction. The event's action represents the result of executing the corresponding instruction. Since the JVM specification essentially addresses type checking, the Event-B specification abstracts from the actual value of bytecode variables and only models their types. To generate a test, we use the execution traces of this Event-B model, since each event in the trace denotes an instruction. To generate vulnerability tests, we modify the specification of an event to negate its guard, thus representing a violation of the JVM specification, in order to generate traces that denote invalid programs. This approach is modular, easily extensible, and it can reuse existing tools like ProB [15] to generate test cases.

The rest of this paper is structured as follows. Section 2 describes some security issues of Java-based smart cards. Section 3 describes our methodology for generating vulnerability tests. Section 4 reports on the application of our approach to five Java Card bytecode verifiers. Finally, we conclude with an appraisal of our approach and future work in Section 5.

2 Java Card Security Issues

The Java Card platform is a multi-application environment where critical data of an applet must be protected against malicious access from another applet. To enforce protection between applets, classical Java technology uses type verification, class loader and security managers to create private name spaces for applets. In a smart card, complying with the traditional enforcement process is not possible. On the one hand, the type verification is executed outside the card due to memory constraints. On the other hand, the class loader and security managers are replaced by the Java Card firewall.

To allow code to be loaded into the card after post-issuance raises security issues similar to those of web applets. An applet not built by a compiler (handmade bytecode) or modified after the compilation step may break the Java sandbox model. Thus, the client must check that Java typing rules are preserved at the bytecode level. However, an attacker may attempt to build a bytecode program that confuses a return address with an object reference, thus allowing inspection and modification of critical memory values. The absence of pointer operators in the Java programming language reduces the number of programming errors. But it does not stop attempts to break security protections with unfair uses of pointers.

2.1 Logical Attacks in Smart Card

An attack can be carried using an ill–formed applet to obtain sensitive information stored in the card; for obtaining it, the applet will try to execute some illegal instructions to read and write in the smart card memory as explained in [11]. This can be accomplished by making a type confusion attack or by changing the control flow graph. Type confusion blurs the Java Card Runtime Environment to use reference to an object's instance as a value. In Java Card, references are mainly stored as 16 bit, i.e. the size of a short. This attack can be achieved by pushing a value and manipulating it as a reference. There are four methods to obtain a type confusion.

1. Input file manipulation. The goal is to modify the file after the compilation step to bypass the BCV. An on-card BCV will mitigate these attacks. Other BCVs are only partially embedded due to the smart card constraints.
2. Fault injection. This technique injects energy on the chip which is transformed into electric signals, which in turn can change values in memory or let the program behave differently by skipping instructions, inverting results and so on.
3. Shareable interfaces mechanisms abuse. To perform this attack, one creates two applets which communicate using the shareable interface mechanism. To create a type confusion, each of the applets use a different type of array to exchange data and are compiled separately. During the load phase, there is no way for the BCV to detect such a problem.
4. Transaction mechanisms abuse. The purpose of a transaction is to bundle a group of operations together. By definition, the rollback mechanism should also deallocate any objects allocated during an aborted transaction, and reset references to such objects to null. However, sometimes cards keep the references of objects allocated during transaction even after a roll back. Then, allocating a new object allows to point on the same memory segment with two references having two different types.

The first approach is the easiest way to perform logical attack against smart cards. The (3) and (4) are now correctly handled by recent smart cards. The second approach requires specific equipment, but its effects are exactly the same as the first attack and can be partially mitigated with dynamic run time type verification.

2.2 Java Card Byte Code Verifier

The BCV is a complex piece of software, and the algorithms to perform the verification were too expensive both in term of memory and computing requirements to be embedded in a Java Card except in the 3.0.4 *Connected Edition* version. In this version, the BCV is similar to the KVM verifier [17] where the idea is to separate the verification process in two parts: an off-card part, that computes a certificate, or "proof" that the code is correct with respect to the security policy, and an on-card part, that uses the certificate to verify the correctness of downloaded code. The "proof" generated is similar to the *StackMap* attribute used by the KVM, and contains the same kind of information. Due to the fact that no products are available for this platform, we focused on the 3.0.4 *Classic Edition* version, an evolution of the 2.2 version where the BCV is optional.

This section describes the Java byte code verification process that has to be performed. This verification should be performed for each package loaded and should reject the whole package if one of the tests fail. The full description of the verification can be found in [16], and a more detailed description, with appropriate discussions is given in [7]. This last description clarifies most of the unclear or ambiguous parts in the official JVM specification. A difference between Java and Java Card verification is that the verification has to be performed on Converted APplet (CAP) files for Java Card instead of class files. A CAP file is a tokenized and optimized version of a set of Java classes.

First, tests are performed on a CAP file when it is downloaded in order to ensure that the CAP file is well formed. Those tests do not analyze the code, but aim to check that the file is well structured. For example, it checks that no method is empty, or that mandatory parts of the file exist. For example, it is ensured that no final method is overridden, or that no class inherits from one of its subclasses. Moreover, in the case of Java Card, if one of the loaded classes already exists in the card, then the verification should fail.

Then, the type correctness of the program is verified. This verification is performed on a method basis, and has to be done for each method present in the package. When a method is invoked, a *frame* is created on top of the *Java virtual machine stack*. A frame contains the method's local variables and an *operand stack* which is used to store intermediate results during method execution. The size of the operand stack of a method is determined at compile time. JBC instructions play with these variables and the operand stack. A frame state denotes the value of the local variables and the operand stack.

Type checking ensures that no disallowed type conversions are performed. For example, an integer cannot be converted into an object reference, downcasting can only be performed using the `checkcast` instruction, and arguments provided to methods have to be of compatible types. Since the types of the local variables is not explicitly stored in the bytecode, they are derived by analysing the bytecode. This part of the verification is the most complicated one, and is both time and memory expensive. It requires computing the type of each variable and stack element for each instruction and each execution path. In order to make such verification possible, the verification is quite conservative on the programs that are accepted. The standard bytecode verification algorithm only accepts programs where the type of each element in the stack and local variable is the same, whatever the path taken to reach an instruction. This also requires that the size of the stack is the same for each instruction for each path that can reach this instruction. Additionally, as every method defines the maximum size that the stack can take during execution, it is checked so that neither overflow or underflow can occur.

Here is an excerpt of the JVM specification for instruction `sload x`, which loads a short from the local variable identified by index `x` in the frame of a method invocation.

Stack
...→
..., `value`
Description
The index is an unsigned byte that must be a valid index into the local variables of the current frame [...]. The local variable at index must contain a short. The value in the local variable at index is pushed onto the operand stack.

The description section provides the precondition and the postcondition of the instruction, while the stack section describes how the operand stack is modified by the instruction and the element required on the top of the stack before execution.

2.3 Verifying the Verifier

Cohen [2] has done a preliminary work on verifying the correctness and proposed a complete formalization for defensive JVM using ACL2. Each instruction in this model consists of operational semantics that describes its behaviors and also the static constraints that express the conditions needed to execute the instruction. Stata et Abadi [21] were the first to use typing rules to model the BCV. These rules precise the behavior of the instructions, describing the inputs, the execution context and all the postconditions of each instruction on the context. Freund et Mitchell [9] used the same framework to evaluate the object initialization considering only a minimum set of instructions. They added other Java features like classes, interfaces, arrays and exception in [10] and they proved the correctness of their type system.

Considering a set of important Java instructions, Qian [20] achieved one of the most complete works which proved the correct execution of a program by verifying its type system. He also proposed a proof for the verifier which is extracted from its formal model. In [6], a correct implementation of a BCV was explained by considering the verification problem as a data flow analysis and the executable was extracted using the Specware tool. Push *et al* [19] in the project named Bali used the Isabelle/HOL prover to define and verify the properties on subset of Java called μ-java. They formalized Qian's type system and its semantics. Nipkow in [13] has modeled a complete Java BCV using Isabelle.

His idea was to provide a generic proof for the verification algorithm and to instantiate it for a particular VM. The specific verification algorithm exploiting the *StackMap* attribute has been proved correct using its complete formalization and its proof in Isabelle [12].

In 1998 Gemplus demonstrated the correctness of Java Card optimizations availing B method [14]. Deutsche Telekom [3] employed a model checker (SMV) to demonstrate the Java Card verification algorithm which was realized using Linear Temporal Logic. A similar approach by applying the SMV model checker was used by Gemplus [4] to ensure whether the confidentiality of a shared data was preserved for a given applet using a causal dependency model. The first smart card using synthesized code from formal specifications was exhibited at Java One by Gemplus in 2002.

From all these studies, it is obvious that such a piece of standardized code and its implementation should be correct. However, Thales ITSEF [8] reported at the Common Criteria conference in 2010 a bug that allows a type confusion in a Java Card. The specification of a verifier changes very rarely, but Java Card is an exception with on-card verifiers. As high-level optimization is required, some differences may be expected.

There are very few implementations of verifiers that are publicly used. We assume that Oracle's verifier is the most commonly used, even if each smart card manufacturer has developed its own optimized version. Testing a BCV is a hard task. Static code analysis tool are used, but they are not easy to use due to the level of abstraction required. So there still exists an issue with both Java Card editions. The correctness of a particular implementation of the bytecode verifier needs at least a test suite or a methodology to check its correctness.

3 Methodology for Generating Vulnerability Tests

Our goal is to generate vulnerability tests for the BCV. We proceed as follows. A BCV vulnerability test is a faulty bytecode program. A bytecode program is a sequence of bytecode instructions. To generate a bytecode program, one can build a formal model of the bytecode language where each operation denotes a bytecode instruction. An execution trace of such a model then denotes a bytecode program. A faulty bytecode program contains an instruction which can be executed when its precondition is false. Thus, to generate a faulty bytecode program, one simply has to negate the precondition of an instruction and try to execute it in an execution trace. We use the Event-B notation to represent our formal model. In Event-B, operations are called "event", so we will use that term in the sequel.

In order to verify our approach, we have selected a subset of twelve instructions of the Java bytecode language (`aconst_null`, `pop`, `return`, `sadd`, `sconst_n1`, `sconst_0`, `sconst_1`, `sconst_2`, `sconst_3`, `sconst_4`, `sconst_5`, `sload`). These instructions manage the operand stack and the local variables. Our model can be used for testing stack overflows and underflows, type confusion on primitive types, for both local variables and stack elements.

For example, an accepted test could be the following sequence (where local variable at index 4 is a *short*): [`sload_4; sconst_2; sadd; return`].
A rejected test could be: [`sconst_2; aconst_null; sadd; return`].

3.1 The Formal Model

Figure 1 represents the variables and the invariants of the Event-B model. Variable *pc* denotes the index of the next instruction to be generated. The frame state of an instruction is represented by variables *s* and *v*, which respectively denote the operand stack and the local variables associated to each instruction of the bytecode program to be generated, whose length is given by constant *maxpc*. Thus, we store a copy of the "before" frame state for each instruction. It allows us to generate test cases for branching instructions. Branching instructions entails that an instruction can be reach from several execution paths. In a valid bytecode program, all frame states resulting from an execution path to an instruction must be type compatible, so that whatever path is taken, an instruction is always executed with valid types. Constant *maxstack* denotes the maximum size of the operand stack, which is determined during compile time. The size of the stack $s(i)$ is given by $z(i)$; this variable is necessary to generate faulty instructions for stack underflows. Variable *halt* is set to true when the program has reached a valid frame state for completing a method.

INVARIANTS

inv1 : $\mathtt{pc} \in 1 \mathinner{..} \mathtt{maxpc}$
inv2 : $\mathtt{s} \in 1 \mathinner{..} \mathtt{maxpc} \pfun (0 \mathinner{..} \mathtt{maxstack} - 1 \pfun \mathtt{TYPE})$
inv3 : $\mathtt{v} \in 1 \mathinner{..} \mathtt{maxpc} \pfun (1 \mathinner{..} \mathtt{maxlocalvar} \pfun \mathtt{TYPE})$
inv4 : $\mathtt{z} \in 1 \mathinner{..} \mathtt{maxpc} \pfun 0 \mathinner{..} \mathtt{maxstack}$
inv5 : $\mathtt{halt} \in \mathtt{BOOL}$
inv7 : $\mathtt{dom(s)} = \mathtt{dom(v)} \land \mathtt{dom(s)} = \mathtt{dom(z)} \land \mathtt{dom(v)} = \mathtt{dom(z)}$

Fig. 1. Invariants

Figure 2 represents the *Initialisation* event. The initial state of the machine contains, for instruction 1, an empty stack and some local variables defined by constant *initlocalvar*; the frame states of the other instructions are undefined.

The model of instruction `sload` is given in Figure3. Guards `grd1_t`, `grd2_t` and `grd3_t` represent the security conditions defined in the Java Specification [16]. Guards `grd1` and `grd2` have been added to control the test generation process. Guards with suffix "`_t`" will be negated to generate test cases; guards without suffix "`_t`" are never negated. Guard `grd1` ensures that no event can be executed after variable `halt` has been set to `TRUE`; variable `halt` is set to `TRUE` by instruction `return`, which ends each bytecode execution for a method. Guard `grd2` ensures that the length of execution traces does not exceed `maxpc`. Stack overflow is controlled by guard `grd1_t`. Guard `grd2_t` checks that the index is a valid index of the array of local variables. Guard `grd3_t` checks that the type of the local variable at the given index is a short. The event actions modify the

```
EVENTS
Initialisation
    begin
        act1 : pc := 1
        act2 : s := {1 ↦ ∅}
        act3 : v := {1 ↦ initlocalvar}
        act4 : z := {1 ↦ 0}
        act6 : halt := FALSE
    end
```

Fig. 2. Initialisation

frame state of the next instruction. Thus, a short is pushed onto the stack (`act2` and `act3`) and the local variables are left unchanged (`act4`).

```
EVENTS
Event  sload ≙
    any
        index
    where
        grd1 : halt = FALSE
        grd2 : pc < maxpc
        grd1_t : z(pc) ≤ maxstack − 1
        grd2_t : index ∈ 1 .. maxlocalvar
        grd3_t : v(pc)(index) = short
    then
        act1 : pc := pc + 1
        act2 : s := s ⩤ {pc + 1 ↦ s(pc) ⩤ {z(pc) ↦ short}}
        act3 : z := z ⩤ {pc + 1 ↦ z(pc) + 1}
        act4 : v := v ⩤ {pc + 1 ↦ v(pc)}
    end
```

Fig. 3. `sload` event

3.2 The Test Generation Process

To obtain the test suite using MBT, the process is split into 3 steps as illustrated in Figure 4: test generation, concretization and execution. We have developed a tool for each step. Abstract test generation is performed by the Vulnerability Tests Generator (VTG). Then the XML2CAP tool translates these abstract tests to CAP files. Each CAP file is a concrete test. Finally TestOnPC and TestOnCard execute the tests on the off-card part and on-card part, respectively.

VTG, depicted in Figure 5, is the test generator. Compared to a traditional MBT approach, VTG includes a new step, faulty model derivation, between the model and the test generation. This new step generates a set of faulty models from the original model.

Tests can be split in 3 parts. The preamble leads the system under test (SUT) from the initial state to a state where it is possible to execute the body. The body is the execution of the tested behavior. Finally the postamble leads the SUT to a desired state.

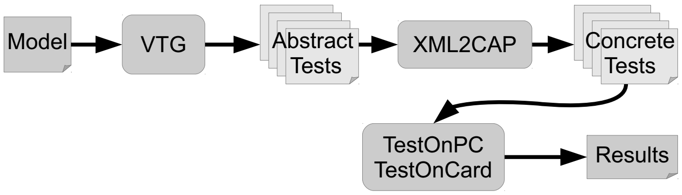


Fig. 4. Our MBT process

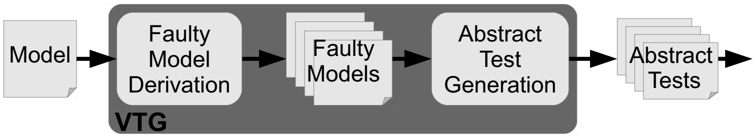


Fig. 5. VTG process

3.3 Faulty Model Derivation

Algorithm 1 describes the faulty model derivation process. Each generated faulty model contains only one faulty instruction, in order to ease the location of faults in the BCV. A faulty instruction is a negation of the JVM specification preconditions. Since there are several ways of negating a condition, several models are produced for a single faulty instruction. The faulty model contains a new state variable, `eut` (event under test) which ensures that a faulty instruction in executed only once in a test. Variable `eut` is initialized to `FALSE` and set to `TRUE` by the execution of the faulty instruction; a guard "`eut` = `FALSE`" is added to the faulty instruction so that it is executed only once.

To negate a guard g, the algorithm uses function $neg(g)$, which returns the set of negations of guard g. It is computed by recursively applying *derivation rules*. A negation g' of a guard g satisfies the following property: $g' \Rightarrow \neg g$. Thus, there are several possible negations g' for a guard g. The negations we consider are defined by derivation rules. The derivation rules necessary to rewrite the instructions of our subset of the Java bytecode language are presented in Figure 6. A rule has the following form: $neg(f) \leadsto \{f_1, \ldots, f_n\}$. Each f_i is a negation of f, and it may include a call to neg as a subformula. Thus, these rules are applied recursively until no more neg appear. Termination is ensured by (manually) checking that the rules decrease the height of the formula's abstract syntax tree. Completeness is manually checked by ensuring that $\neg f \Leftrightarrow f_1 \vee \ldots \vee f_n$. These are proved using the prover of the Rodin toolkit, which supports the Event-B method.

```
Input: $m$ : Event-B model
Output: $M'$ : set of Event-B model
for each event $e$ of $m$ do
    rewrite the guard of $e$ into two guards:
        $grd$, the conjunction of all guards of $e$ without suffix "_t";
        $grd\_t$, the conjunction of all guards of $e$ with suffix "_t";
end
for each event $e$ of $m$ do
    $e.RW := neg(grd\_t)$;
end
for each event $e$ of $m$ do
    for each $rw$ in $e.RW$ do
        add a new model $m'$ to $M'$ such that
            $m' := m$;
            $m'.events := m'.events \cup \{e'\}$, where $e'$ is defined as follows:
                $e' := e$;
                replace $e'.grd\_t$ by $rw$;
                add guard "eut = FALSE" to $e'$;
                add action "eut := TRUE" to $e'$;
    end
end
```

Algorithm 1. Faulty model derivation algorithm

Using derivation rules provides flexibility for controlling the faulty model generation process. For instance, rule 1 describes that a conjunction can be negated in three different ways: exactly one of the conjunct is false or both conjuncts are false. Some rules are also specific to a problem domain. For instance, to negate a formula $h(a) = b$, one may want to test two cases, instead of using rule 4 of Figure 6: i) the case where $h(a)$ is undefined (*i.e.,* $a \notin dom(h)$) and the case where $h(a)$ is defined. This would be represented by the following rule:

$$4'. \quad neg(h(a) = b) \rightsquigarrow \{a \notin dom(h),\ a \in dom(h) \land a \mapsto y \notin h\}$$

The application of derivation rules to a guard may generate a predicate which is unsatisfiable, or there may not exist a state reachable from the initial state that satisfies the generated predicate. These cases are detected during the abstract test generation step, which involves a model checker. In our tool, the user can check the list of generated predicates and delete those which are obviously not satisfiable, in order to speed up the test generation step.

As an example of applying derivation rules, consider instruction `sload`. Its guard to negate (*i.e.,* the conjunction of guards with suffix "`_t`" in Figure 3) is the following:

$$(z(pc) \leq maxstack - 1) \land (index \in 1\,..\,maxlocalvar) \land (v(pc)(index) = short)$$

Applying the rules of Figure 6, we obtain the following negations; elements identified in red highlight the modified part of the original guard.

1. $neg(p_1 \wedge p_2) \rightsquigarrow \{neg(p_1) \wedge p_2,\ p_1 \wedge neg(p_2),\ neg(p_1) \wedge neg(p_2)\}$
2. $neg(i_1 \leq i_2) \rightsquigarrow \{i_1 > i_2\}$
3. $neg(i_1 \geq i_2) \rightsquigarrow \{i_1 < i_2\}$
4. $neg(i_1 = i_2) \rightsquigarrow \{i_1 \neq i_2\}$
5. $neg(a \in B) \rightsquigarrow \{a \notin B\}$

Fig. 6. Relevant derivation rules

1. $z(pc) > maxstack - 1 \wedge index \in 1 \mathinner{..} maxlocalvar \wedge v(pc)(index) = short$
2. $z(pc) \leq maxstack - 1 \wedge index \notin 1 \mathinner{..} maxlocalvar \wedge v(pc)(index) = short$
3. $z(pc) > maxstack - 1 \wedge index \notin 1 \mathinner{..} maxlocalvar \wedge v(pc)(index) = short$
4. $z(pc) \leq maxstack - 1 \wedge index \in 1 \mathinner{..} maxlocalvar \wedge v(pc)(index) \neq short$
5. $z(pc) > maxstack - 1 \wedge index \in 1 \mathinner{..} maxlocalvar \wedge v(pc)(index) \neq short$
6. $z(pc) \leq maxstack - 1 \wedge index \notin 1 \mathinner{..} maxlocalvar \wedge v(pc)(index) \neq short$
7. $z(pc) > maxstack - 1 \wedge index \notin 1 \mathinner{..} maxlocalvar \wedge v(pc)(index) \neq short$

Negations 2, 3, 6 and 7 are unsatisfiable, because of the conjunction $index \notin 1 \mathinner{..} maxlocalvar \wedge v(pc)(index) = short$. When $index \notin 1 \mathinner{..} maxlocalvar$ holds, expression $v(pc)(index)$ is undefined. Figure 7 illustrates a faulty instruction obtained with negation 5.

```
EVENTS
Event  evt_sload_11_24_EUT ≙
    any
        ...
    where
        grd :  halt = FALSE ∧ pc < maxpc
        grd_t :  z(pc) > maxstack − 1 ∧ index ∈ 1 .. maxlocalvar ∧
                 ¬v(pc)(index) = short
        grd_EUT :  eut = FALSE
    then
        ...
        act_EUT :  eut := TRUE
    end
```

Fig. 7. `evt_sload_11_24_EUT` event

3.4 Abstract Tests Generation

With these new models, we generate tests. We use ProB [15] to find traces containing the preamble, the body and the postamble. This search is driven by two parameters. First we specify the depth. This parameter represents the maximum length of the traces *i.e.,* the maximum number of events for a test. The second parameter is a predicate that each trace must satisfy. Presence of the EUT and final state are represented by the predicate $eut = TRUE \wedge halt = TRUE$. Other parameters must be specified; the reader is referred to the ProB website [1] for more details.

For each model, we only generate a subset of possible traces containing the EUT. For each transition, when several solutions can satisfy the guard of an event, only one solution is used. The first solution found for the model of Figure 7 is :

- preamble: `INITIALISATION; aconst_null; aconst_null; aconst_null;`
- body: `evt_sload_11_24_EUT`;
- postamble: `return` .

At the initialization, the local variables contain two references and the maximum size of stack is equal to 3. The execution of three events *aconst_null* fills the stack. In this state, the faulty event can be executed. Finally the return event leads the machine to the end of the test.

3.5 Concrete Tests

A concrete test is a CAP file which contains a class with one method. This method contains the instruction trace generated with ProB. The XML2CAP tool generates one CAP file for each trace. It computes, among other things, the maximum stack size. Because several other informations are required for composing a CAP file, we use a predetermined CAP file which is completed with the generated method. The local variables are fixed in the predetermined CAP file and in the abstract model.

4 Evaluation

Tests Computation. Our experiments have been performed on a MacBook Pro with a 2,3 GHz Core i5 dual-core processor, 8GB of RAM and a 5400 $t.m^{-1}$ hard disk with 8MB of cache. Each measure has been made three times and we provide the average.

The first step is the negation of guards. It processes four distinct guards. It generates sixteen negations in less than one second. Eight of the negations are unreachable and we only keep the eight reachable models. The second step is the model derivation. We produce eighteen new models in twenty seconds. The last step is the abstract test generation. The results obtained vary depending on the search depth.

Table 1 represents the results we obtain for the generation of abstract tests. The first column represent the depth parameter. The penultimate line of this table represents the results we obtain if we do not remove the unreachable negations. In the last line we take the shortest depth for each model. The second and third columns represent respectively the time and the number of tests extracted. The next column represent the time taken in average to generate one test. The penultimate column provides the percentage of models which can produce at least one test for the given depth (the depth may not be large enough to generate a test). The last column represents the test coverage with respect to a test plan manually derived by an expert. In this manual test plan, the expert has identified for each instruction a number of test cases. The test cases considered by our approach depend on the derivation rules. We have computed the number of manual test cases that our approach can reproduce using the derivation rules. Because we only work on a subset of the Java Card instruction set, the manual test cases which involve other instructions could not be reproduced; thus the last column is never equal to 100%.

Table 1. Abstract test generation evaluation

Depth	Time	Nb of tests	Speed (*sec/test*)	Model coverage	Manual plan coverage
3	1min30	30	3,0	13%	12%
4	12min30	432	1,7	33%	31%
5	1h30	10133	0,5	100%	93%
5 Full	2h30	10133	0,9	65%	93%
*	1h05	7318	0,5	100%	93%

Smart Card Execution. After the execution of the vulnerability test suites, we classify the results as: i) accepted and correctly executed CAP file, ii) rejected CAP file, and iii) accepted CAP but rejected executions. An accepted CAP means that the embedded load phase verifications do not detect the ill formed file. An accepted CAP but rejected during execution detects the presence of a run time check. The test suites have been evaluated on five different smart cards from two different smart card manufacturers ($\{a, e\}, \{b, c, d\}$). Cards a, b, c are Java Card 2.1 while d and e are Java Card 2.2 standard.

Card d allows the execution of all tests. Thus, it is possible to generate stack over and under flow. It does not mean that the vulnerabilities can be exploited. To obtain an executable attack, one often needs several vulnerabilities. For example, if the card does not check the local variable it offers the possibility to change the return address. But the return address can be protected by an integrity check. With our test results, one may be able to characterize a given implementation and then provide information to set up an attack.

For the other cards, some tests fail (the card become mute) during the execution or during the load. On card a, we have been able to find a chain of vulnerabilities that allows us to execute a shell code on the card.

Overall Effort for the Complete Process. We estimate that it would take a fresh person about 25 hours to build the formal model of the subset of twelve instructions. We could not precisely measure this effort, because our model is the result of several iterations on different subsets of similar bytecode languages (for instance, our first experiments were conducted on the language of Freund and Mitchell [10]). The manual removal of unreachable guards takes about 1.5 hour. The model derivation step does not require any human intervention. We then must choose appropriate parameters for the abstract test generator step. This will only take a few minutes. Then we launch the abstract test generator.

For a full coverage, this step takes 1.5 hour. Finally we concretize the tests in a few minutes. Overall, the full process for the twelve instruction subset takes about a week.

It took eight person-months to develop the complete toolchain. Many parts of this toolchain are re-usable. Only the concretization and the execution tools must be adapted for a new problem.

Manually writing a test suite for the BCV is tedious. It took us one week to manually derive a test suite for our subset of twelve instructions. Moreover, this test suite contains only one test per test case. Our solution can produce all possible tests in roughly the same time; we have controlled the number of tests using a timeout. Our vulnerability coverage is slightly less, but our tests are more complex and they test vulnerabilities in many contexts. When the full bytecode instruction set will be tackled, we expect increased productivity gains, because automation will easily generate a greater variety of contexts for testing an instruction.

5 Conclusion and Future Work

We have proposed a new methodology to generate vulnerability tests and developed several tools supporting it. Our method is based on using a formal model of the system under test which represents security constraints. A standard MBT approach is applied and we obtain a set of vulnerability tests. We applied our methodology to the testing of five Java smart card BCV. We have discovered vulnerabilities in all of them. Our approach can be used by a certification authority or an evaluation center in order to set up vulnerability analysis. This would ease the characterization of the embedded software.

The Java Card byte code verifier is a key component in the security of Java-based smart cards and finding weaknesses is of prime importance. Our experiment with a subset of the Java Card instruction set constitutes a proof of concept. We plan to apply our methodology to the complete Java Card instruction set, including the type lattice, the subroutine mechanism and the exception mechanism.

Our methodology is generic and can be applied to other security components. We have started to model the payment protocol EMV, which includes cryptographic primitives, in other to generate vulnerability tests.

References

1. `http://www.stups.uni-duesseldorf.de/ProB`
2. `http://www.computationallogic.com/software/djvm/`
3. Basin, D., Friedrich, S., Posegga, J., Vogt, H.: Java bytecode verification by model checking. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 491–494. Springer, Heidelberg (1999)
4. Bieber, P., Cazin, J., Girard, P., Lanet, J.L., Wiels, V., Zanon, G.: Checking secure interactions of smart card applets: Extended version. Journal of Computer Security 10(4), 369–398 (2002)

5. Casset, L.: Development of an embedded verifier for java card byte code using formal methods. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 290–309. Springer, Heidelberg (2002)
6. Coglio, A., Goldberg, A., Qian, Z.: Toward a provably-correct implementation of the JVM bytecode verifier. In: Proc. OOPSLA 1998 Workshop on Formal Underpinnings of Java, pp. 403–410 (1998)
7. Doyon, S.: On the security of Java: The Java Bytecode Verifier. Master's thesis, Université Laval, Québec City, Canada (1999)
8. Faugeron, E.: How to hoax an off-card verifier. In: e-Smart, Sophia Antipolis, France, September 21-24. Strategies Telecoms & Multimedia, pp. 310–328 (2010)
9. Freund, S.N., Mitchell, J.C.: A type system for object initialization in the Java bytecode language. In: Freeman-Benson, B.N., Chambers, C. (eds.) ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, pp. 310–327. ACM Press (1998)
10. Freund, S.N., Mitchell, J.C.: A formal framework for the Java bytecode language and verifier. In: Hailpern, B., Northrop, L.M., Berman, A.M. (eds.) ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, pp. 147–166. ACM Press (1999)
11. Iguchi-Cartigny, J., Lanet, J.: Developing a Trojan applet in a Smart Card. Journal in Computer Virology 6, 343–351 (2010)
12. Klein, G., Nipkow, T.: Verified lightweight bytecode verification. Concurrency and Computation: Practice and Experience 13(13), 1133–1151 (2001)
13. Klein, G., Nipkow, T.: Verified bytecode verifiers. Theor. Comput. Sci. 3(298), 583–626 (2003)
14. Lanet, J.L., Requet, A.: Formal proof of smart card applets correctness. In: Schneier, B., Quisquater, J.-J. (eds.) CARDIS 1998. LNCS, vol. 1820, pp. 14–16. Springer, Heidelberg (2000)
15. Leuschel, M., Butler, M.: ProB: An automated analysis toolset for the B method. International Journal on Software Tools for Technology Transfer 10(2), 185–203 (2008)
16. Lindholm, T., Yellin, F.: Java Virtual Machine Specification, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
17. Sun Microsystems: Connected, limited device configuration, specification 1.0a, Java 2 platform micro edition (2000)
18. Sun Microsystems: Virtual machine specification Java Card platform (May 2009), `http://www.oracle.com`
19. Pusch, C.: Proving the soundness of a Java bytecode verifier specification in isabelle/HOL. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 89–103. Springer, Heidelberg (1999)
20. Qian, Z.: A formal specification of java-TM virtual machine instructions for objects, methods and subroutines. In: Alves-Foss, J. (ed.) Formal Syntax and Semantics of Java. LNCS, vol. 1523, pp. 271–312. Springer, Heidelberg (1999)
21. Stata, R., Abadi, M.: A type system for Java bytecode subroutines. In: MacQueen, D.B., Cardelli, L. (eds.) Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1998, San Diego, CA, USA, January 19-21, pp. 149–160. ACM (1998)

Integrating Formal Predictions of Interactive System Behaviour with User Evaluation

Rimvydas Rukšėnas[1], Paul Curzon[1], and Michael D. Harrison[1,2]

[1] School of Electronic Engineering and Computer Science, Queen Mary University of London
{r.ruksenas,paul.curzon}@eecs.qmul.ac.uk
[2] School of Computing Science, Newcastle University, Newcastle upon Tyne, UK
michael.harrison@newcastle.ac.uk

Abstract. It is well known that human error in the use of interactive devices can have severe safety or business consequences. It is important therefore that aspects of the design that compromise the usability of a device can be predicted before deployment. A range of techniques have been developed for identifying potential usability problems including laboratory based experiments with prototypes and paper based evaluation techniques. This paper proposes a framework that integrates experimental techniques with formal models of the device, along with assumptions about how the device will be used. Abstract models of prototype designs and use assumptions are analysed using model checking techniques. As a result of the analysis hypotheses are formulated about how a design will fail in terms of its usability. These hypotheses are then used in an experimental environment with potential users to test the predictions. Formal methods are therefore integrated with laboratory based user evaluation to give increased confidence in the results of the usability evaluation process. The approach is illustrated by exploring the design of an IV infusion pump designed for use in a hospital context.

1 Introduction

Experiments in usability laboratories play an important role in providing understanding of the way people behave when using interactive devices in specific circumstances. These experiments can be used to identify flaws in the design of devices before deploying them in a wider context. Experiments, by their nature, are not exhaustive with respect to behaviour and so some behaviour that would occur under the assumptions of use made, may not be observed within the confines of the experiment. It is also difficult to predict all possible interactions that may lead to a particular observed behaviour and so ought to be dealt with in the interaction design. This is particularly difficult when investigating human error. For example, Vicente et al [1] have estimated the rate of errors related to number entry tasks, such as those arising from programming medical infusion devices, to be in the range of 1 in 33,000 to 1 in 338,800 for an example device. In experiments error rates have to be increased artificially by, for example, adding secondary tasks to increase working memory load to overcome such problems. This can lead to those experiments not being ecologically valid: the errors seen may not actually correspond to real situations.

This paper describes a framework that can be used to highlight error prone interaction design given a set of cognitive assumptions. When evaluating an interactive system design it is necessary not only to describe assumptions about the device design but also the assumptions that are being made about the user in terms of capabilities and context. An approach to exploring the consequences of such assumptions is proposed that combines formal verification techniques with laboratory-based experiment. We claim that this approach gives increased analytical power to experimental results. The behavioural assumptions and configuration of an experiment including the device are modelled at a high level of abstraction. This allows model checking to be used to explore the consequences of the behavioural assumptions exhaustively in a way that is not possible using experimental or simulation techniques. Formal methods are therefore integrated with laboratory based user evaluation to give increased confidence in the results of the usability evaluation process.

The approach contrasts but also has some parallels with the use of cognitive modelling to analyse user assumptions [2], in particular the use of cognitive architectures. A base set of assumptions is used, similar to cognitive architectures, that are relatively independent of the task. However cognitive architectures provide much more detailed models of cognitive processes such as visual or auditory perception, memory and learning, whereas the approach described here takes a more abstract view of these processes as discussed in more detail in Section 3.2.

Another important difference relates to the way the models are used to carry out analysis. In cognitive modelling approaches, the analysis of system properties is based on individual simulation runs of a relatively small number of possible behavioural traces. The idea is that each trace represents statistically 'average' or 'likely' behaviour. The result is that models which are effectively deterministic are used to predict likely properties of an interactive system. In the approach described here the models are more abstract and involve a high degree of non-determinism. This non-determinism generates a wide range of behaviours. It allows *exhaustive* exploration of the consequences of the modelled assumptions that lead to these behaviours using automatic tools such as model checkers.

In addition to testing predictions based on the models the analysis gives insight about the design of the interactive device and helps the evaluator to consider the validity of the experimental design. It therefore provides insight into the confidence with which the results of the experiment and its interpretation can be considered. It can highlight mismatches between the consequences of the assumptions and experimental results and so lead to suggestions of further experiments as well as ruling out potential explanations for those mismatches.

To illustrate the proposed approach we explore a design that supports the access of information within two tasks. The tasks can either be interleaved or carried out sequentially. Our aim is to consider the effectiveness of the design given a set of cognitive assumptions, in particular the soft constraints hypothesis [3]. Modelling predicted that errors would be made that matched those observed in the experiment. However the specific traces identified by the model checker that led to errors were different to the sequence of actions followed by participants in the experiment. In particular the formal analysis predicted a different form of interleaving from that assumed or seen in the experiment. This suggests that the modelled assumptions are not sufficient to fully explain the observed behaviour. Possible explanations for why the actual behaviour differs from that derived from the assumptions could be explored both by further experiment and/or by modelling them and model checking to derive the consequences.

The contribution of this paper is therefore to:

- present a novel way of combining formal reasoning technology and laboratory-based experiments to explore the consequences of the design of an interactive environment, and
- demonstrate on a specific example how the technique can be used to give insight into experimental results and in particular make explicit the behavioural assumptions made and its design consequences.

2 Related Work

The use of formal modelling and analysis as a means of developing hypotheses for experimental evaluation of interactive systems appears to be a novel approach. There is however a significant literature on combinations of user or use assumptions with models of devices. These range from assumptions based on task models to assumptions based on cognitive models. A recent example of the former approach is the work of Bolton and others [4] which also contains a good review of related material. They use the Enhanced Operator Function Model (EOFM) to describe operator tasks. This task model is combined with a device model as a basis for a model checking analysis using SAL [5]. The analysis involves considering variants of the task by inserting automatically "phenotypes of erroneous human behaviour. These variant models are checked against correctness properties - that the combined model would reach specified goals. Observable manifestations of erroneous behaviour are also explicitly modelled by Fields [6] who also analysed error patterns using model checking. Both approaches, however, whilst giving specific kinds of errors to explore in the form of the mistake model lack discrimination between random and systematic errors. The also assume implicitly that there is a correct plan, from which deviations are errors. Beckert and Beuster [7] on the other hand take a step towards combining GOMS modelling and correctness verification. They present a verification environment with a structure similar to the models described here — connecting a device specification, a user assumption module and a user action module, the latter being based on CMN-GOMS. The selection rules of their GOMS model are driven by the assumption model while the actions drive the device model. This gives a way of exploring the effect of erroneous user behaviour in the form of incorrect selection decisions as specified in the user assumption module. However, the assumption module has no specific structure and, thus, does not provide systematic guidance as to what kind of potential errors to explore. These decisions are left to the analysts of the system.

Other relevant research concerns the use of more general assumptions about cognition. A similar approach to the one that forms the basis for this paper is taken by Bowman and Faconti [8]. They formalise one model of human information processing (Interactive Cognitive Subsystems [9]) using the process calculus LOTOS, and then apply a temporal interval logic to analyse constraints on the information flow and transformation between the different cognitive subsystems. Their approach is more detailed than the one described in this paper. It focuses on reasoning about multi-modal interfaces and analyses whether interfaces based on several simultaneous modes of interaction are compatible with the capabilities of human cognition. One source of of user error is cognitive mismatch between user beliefs about the system state or behaviour and the reality. Rushby et al [10] focus on mode errors resulting from cognitive mismatch and the ability of pilots to track mode changes. They formalise a mental model of the system that is specific to the example being considered and then analyse it using the Murϕ verification tool. Their models make no explicit appeal to cognitive principles.

In our earlier work [12], a similar integration of formal verification and laboratory based experiments is employed to provide cognitively grounded accounts of interactive user behaviour. That work, however, does not attempt to evaluate device designs.

3 The Modelling Framework

The proposed analysis of an interactive system is based on a combination of a device model and a user model that together capture assumptions about the design relevant to the context of use. The level of abstraction used for the device specification is determined by the issues under investigation. In the example described here the issue is whether certain task steps are prone to omission given specific design assumptions, not about precise details as to whether a particular task step is carried out. For this reason the device specification is given a high level of abstraction. The specification of plausible user behaviours then follows the same abstraction level. The models are developed using the SAL verification environment [5].

3.1 The Device Model

Our example involves the programming of infusion pumps. Infusion pumps are used both in hospital settings and at home to provide intravenous infusions. They are safety-critical devices, since infusing a drug at the wrong rate or volume may seriously harm patients. As such they provide a realistic test of the viability of the approach.

Analysis is concerned with the task of programming a pump with the prescribed infusion parameters and then commencing the infusion. However because simultaneous programming of two infusion pumps is a common activity in operating theatres, the focus is to consider this multitasking activity and designs where the setting of a pump requires entry of two infusion parameters: the volume to be infused (VTBI) and the duration (time) of infusion[1]. Different makes of infusion pump provide different mechanisms for entering these numeric values. The concern here is not with the details of number entry. This level of abstraction of the pump model captures the generic characteristics of a range of models of infusion pump. The general insights provided by the analysis are therefore likely to be associated with the design characteristic of these different pumps.

Pump operation. The first step in developing the device model is to describe the interactive aspects of pump operation. When the pump is switched on it goes through a setup procedure. Since this step is not a concern of the example, the model simply assumes that the initial state of the pump is on when the setup has finished. The programming

[1] The model would be similar if the rate of infusion is required instead of duration.

options available to the user are presented at a main menu on the pump display on completion of setup. Setting the VTBI and setting the duration are two options which, when selected, move the pump into a mode where the relevant numerical value can be entered. The entered value must be confirmed by pressing the confirmation key. If confirmation occurs when both values have been entered the pump calculates the infusion rate automatically. Pressing the confirmation key returns the pump to the mode where the main menu is displayed. Infusion can then be started using the appropriate key. However, before that, a roller clamp on the pump must be opened.

Pump model. The SAL model of the interactive behaviour for this pump is given in Fig. 1. The variables `vtbi`, `time` and `rate` represent the values of the the infusion parameters. The analysis is to be carried out at a level of abstraction where these numeric values are irrelevant. For this reason the three variables have boolean type. The value `true` indicates that a numeric value for the corresponding infusion parameter has been entered, whereas `false` indicates the opposite. Depending on its mode, the pump shows (some of) these infusion values on the display. The boolean variables `vtbiDisp`, `timeDisp` and `rateDisp` indicate whether the corresponding value is displayed or not. Finally, the boolean `clamp` specifies whether the roller clamp is closed (`true`) or open (`false`).

The mode of the pump operation is specified by the variable `mode`. Three modes of operation are assumed defined as an enumerated type, `Mode`:

```
Mode: type = { off, hold, infusing };
```

The modes `off` and `infusing` indicate that the pump is switched off and infusing, respectively. The `hold` mode represents the remaining states of the pump, when it is switched on but not infusing (for example, being programmed).

The mode of the pump display is specified by the variable `dmode`. The mode can take values defined as the following type `DispMode`:

```
DispMode: type = { dblank, mainmenu, dvtbi, dtime, dinfusing };
```

The value `mainmenu` represents the main pump menu with the programming options presented. The values `dvtbi` and `dtime` represent the numeric entry displays for VTBI and time respectively. The value `dinfusing` represents the display shown during the infusion process. Finally, `dblank` indicates that the display is blank.

The device model of the infusion pump is driven by a set of actions that are defined by input events represented by the following enumerated type `Event`:

```
{ onoff, mvtbi, mtime, enter, confirm, open, close, infuse, tick }
```

At the level of abstraction relevant to the analysis, an input event may correspond to a sequence of button presses. Thus the event `onoff` represents a key press that switches the pump on or off. The events `mvtbi` and `mtime` model users choosing the VTBI and time entry options in the main menu. The events `enter` and `confirm` represent the entry and confirmation of a numeric value (depending on the display mode, this can be either the VTBI or time). Opening and closing of the roller clamp is modelled as the events `open` and `close`. Finally, the time ticking event `tick` represents cases when there is no user action taken.

```
pump: module =
begin
 input event: Event
 local mode: Mode
 output dmode: DispMode, vtbi, rate, time, vtbiDisp, rateDisp, timeDisp, clamp: bool
 initialization
  mode = hold; dmode = dvtbi; vtbi = false; rate = false; time = false; clamp = true;
 definition
  vtbiDisp = dmode = mainmenu or dmode = dvtbi or dmode = dinfusing;
  rateDisp = dmode = mainmenu or dmode = drate or dmode = dinfusing;
  timeDisp = dmode = mainmenu or dmode = dtime or dmode = dinfusing;
 transition
 [  event = onoff and mode = hold --> mode' = off; dmode' = dblank;
 [] event = mvtbi and dmode = mainmenu --> dmode' = dvtbi
 [] event = enter and dmode = dvtbi --> vtbi' = true
 [] event = confirm and dmode = dvtbi --> dmode' = mainmenu; rate' = rate or time;
 [] event = mtime and dmode = mainmenu --> dmode' = dtime
 [] event = enter and dmode = dtime --> time' = true
 [] event = confirm and dmode = dtime --> dmode' = mainmenu; rate' = rate or vtbi;
 [] event = open and clamp --> clamp' = false
 [] event = close and not(clamp) --> clamp' = true
 [] event = infuse and dmode = mainmenu and vtbi and rate and time -->
                  mode' = infusing; dmode' = dinfusing
 [] event = infuse and dmode = dinfusing --> mode' = hold; dmode' = mainmenu
 [] else -->
 ]
end
```

Fig. 1. Pump model in SAL

These events have behaviours (see `transition` section in Fig. 1) that depend on the mode of the device (`mode`). For example, the event `enter` sets `vtbi` to `true` if the display mode is `dvtbi`. This models the entry of the VTBI value when the pump display is in the corresponding mode. In general, events may have the effect of changing two modes: the mode of the device and the mode of the display. They may also change the variables associated with the infusion parameters (`vtbi`, `time` and `rate`).

The example considers the simultaneous programming of two pumps. The model for each pump is derived from `pump` by simple renaming of all variables. For example, `event` is renamed to `events[1]` for the first pump and to `events[2]` for the second. The full device model, `Pumps`, is then defined by composing the pump models.

3.2 The User Model

The purpose of the user model is, when combined with the device model, to restrict the device behaviours to those that are consistent with user behaviour given the cognitive assumptions. The particular user model that is relevant to the analysis of the device is based on an instantiation of an abstract generic user model. These models are generic because they provide the means to replace sets of cognitive assumptions and also because they can be instantiated with the particular task assumptions relevant to the analysis. The model makes it possible for the experimenter to make, and explore, conjectures about use of the interactive system. In this way the approach is not locked into a set of assumptions about how the device will be used.

Flexibility of user models is achieved through three modelling layers. The base layer in the generic model captures core assumptions that are unlikely to be modified by the approach. The intermediate layer, also part of the generic model, specifies the current set of cognitive assumptions. The third layer is an instantiation of the generic model and captures specific assumptions that relate to the details of the device and the task to be performed on it.

The Base Layer. This layer focuses on the mechanisms that relate to the users choice of actions and forms a set of core assumptions. It postulates that actions are chosen non-deterministically but some actions may be preferred to others for cognitive reasons such as their salience. User preferences are modelled using a notion of action *salience* that is informed by the ideas of activation theory [11]. The base layer describes actions and their salience in generic terms. It also specifies termination behaviour, marking the conditions that may lead to a person ending an interaction. More detail on this modelling layer is provided in the earlier paper [12].

The Intermediate Layer. The second (intermediate) layer refines the underlying non-determinism of action choice by introducing salience levels which it uses to partition actions. The notion of salience is also refined by specifying how action salience is derived from associated cues. At this level different assumptions about the salience levels and the relation between salience and cues can be specified. One possible set of such assumptions is described in more detail below and used in the analysis of the example.

The set of assumptions used in the example specify that actions may have two types of cues: sensory (that is external) and internal. The sensory cues are provided by the device and its environment. In the model, they are used to represent any kind (visual, audio, etc.) of external stimulus. The internal cues originate from the user's knowledge of task and device.

Task-knowledge cues can be thought of as a mental representation of the task, and what is necessary to achieve the main task goal. This knowledge is assumed to derive from general training as well as previous experience. The form that task-knowledge cues take in the model is as follows: 'action A $\rightsquigarrow$ action B'. This corresponds to learned behaviour where each action in a sequence provides longer term activation for the next action. Namely, if 'action A $\rightsquigarrow$ action B' and another action C (or series of actions S) are taken instead of B, then the latter still gets activation after execution of C (or S). The second form of task-knowledge cue in the model deals with the first step in a learned series of actions. In this case, there is no preceding action to provide activation and it is assumed that activation may also come from task goals.

The set of *device-knowledge* cues can be thought of as a mental representation of how the device works, and what is necessary to achieve the task using the device. It is assumed that this knowledge derives from repeatedly doing the task on the same device. Thus, the device-knowledge cues as a whole capture learnt sequences of actions. These sequences may also include device-specific actions. This is not the case with the task-knowledge cues. The device-knowledge cues take the following form in the model: 'action A $\rightarrow$ action B'. They represent more procedural aspects of learned behaviour and, consequently, are assumed to provide shorter term activation for the next action in a sequence. Namely, if ‘action A $\rightarrow$ action B’ and another action C is taken immediately after A, then action B ceases to get activation from A.

Action salience and activation levels. The overall salience of an action is determined by the activation provided by its sensory, task-knowledge and device-knowledge cues. The effect of different kinds of cue is assumed to be *equal* and *additive* in nature. Here the equality means that each kind of cue, if present, is assigned a unit (say 1) of activation, while the additivity means that the overall salience of an action is calculated as the sum of these units. Thus, if an action gets activation from all three kinds of cues, then its overall salience will be 3, whereas if it does not have any cues, then the overall salience will be 0. In this set of assumptions, four discrete levels of activation are assumed, each corresponding to one of the possible values (from 0 to 3) of overall salience, so that all the actions are partitioned into these levels. Only actions with the highest level of salience are assumed to be candidates for execution.

Not all user actions that are possible at some point are equally relevant to achieving task goals. For example, the action of starting an infusion is irrelevant when the prescribed infusion parameters have not yet been provided. In the model, the concept of *specificity* refers to the dynamic aspect of cue relevance for such actions. It is assumed that an action being non-specific acts as an inhibitor reducing the activation due to the sensory and task-knowledge cues from 1 to 0. On the other hand, the activation due to the device-knowledge cues is not linked to the specificity of actions in this set of assumptions.

A set of cognitive assumptions like this is chosen on the basis that it is believed to be sufficient to explain behaviour for the given task. If discrepancies with experiments arise then one possibility is that this understanding is incomplete, which in itself is a useful result.

The Concrete Layer. The third (concrete) layer instantiates the generic user model specified by the other two layers to a specific interactive system and its associated tasks. It does this by defining the state space of the user model, the main task goal and the actions and their associated cues specific to the device. In this case the task is programming two infusion pumps.

State space. The state of the user model is specified by the following components: `inp:Inp` giving things the user can perceive in the world, `mem:Memory` giving their beliefs about the state of the system, and `out:Out` giving the actions they can take. The type `Inp` represents assumptions about what the pump users can perceive:

```
Inp: type =
 [# dmode:array [1..2] of DispMode, vtbi:array [1..2] of bool,
   time:..., rate:..., clamp:..., prescription:... #];
```

Here `dmode[i]` ...`clamp[i]` indicate how the user perceives the corresponding attributes on the pump `i`, while `prescription[i]` indicates the perception of the prescription values for the same pump. The type `Memory` represents assumptions about the user’s beliefs about the system state:

```
Memory: type =
   [# pump:[1..2], vtbiSet:array [1..2] of bool,
      timeSet:..., prescription:..., interleave:bool #];
```

Here `pump` indicates which pump is the focus of user attention, `vtbiSet[i]` and `timeSet[i]` represent beliefs as to whether the corresponding infusion parameter has been set, `prescription[i]` represents the memorised prescription values, and `interleave` indicates whether the user has chosen to interleave programming the two pumps or not. Finally, the type `Out` specifies assumptions about which user action (`action`), and on which pump (`pump`), has been chosen by the model:

```
Out: type = [# action: Event, pump: [1..2] #];
```

Task goal. We assume that, from the users point of view, the main goal, `task` for the task of programming the two pumps is to reach a state such that their perception indicates that both pumps are infusing:

```
inp.dmode[1] = dinfusing and inp.dmode[2] = dinfusing;
```

User actions. Programming a pump involves entering the prescribed VTBI first, then confirming it. After that the time option must be selected from the available menu which, as in the case of VTBI, allows entry of the prescribed infusion time (duration) followed by confirmation. The required VTBI and time values can be read from the prescription form. In the experiment the form could have been positioned either nearby or further away from the pump so that the user had two plausible options: to read and memorise both values (VTBI and time) for one infusion, or to consult the prescription form at the time when each of these values had to be entered. When both values have been entered the user is required to open the roller clamp and start the infusion process.

The task description prompts the specification of a set of user actions (as opposed to device actions) in the concrete model layer. The type `ActionNames` defines the names of these user actions:

```
ActionNames: type =
 { memorise, enterVtbi, confirmVtbi, chooseTime,
   enterTime, confirmTime, openClamp, startInfusion };
```

Some of these actions such as `enterVtbi` or `chooseTime` represent groups of key presses. However, these details are deemed to be irrelevant for the analysis of interleaving behaviour and so abstracted away without loss of generality.

These user actions are associated with the action cues as specified in Table 1 (the actual SAL specification is given by defining the relevant parameters for the generic user model). Each cell in this table indicates an action (given by its name) and/or a state condition (written in italic) that is necessary to activate the corresponding cue (given by the column title) for the action given by the row title. For example, the action `enterTime` is cued by the action `enterVtbi` on the task-knowledge level and by the action `chooseTime` on the device-knowledge level. It also has sensory cueing, whereas its specificity (relevance) is defined by the conjunction of the following boolean conditions: "`m.pump` = *this one* OR `m.interleave`" (user is involved in programming this

Table 1. Specification of action cues

Action	Task cues	Device cues	Sensory cues	Specific, if
`memorise`	task goal to start infusion	NONE	NONE	`costs = true` AND `NOT(m.prescription[pump])`
`enterVtbi`	`memorise` OR `costs = false`	`memorise`	YES	(`m.pump` = *this one* OR `m.interleave`) AND `NOT(inp.vtbi[pump])` AND `inp.dmode[pump] = dvtbi`
`confirmVtbi`	NONE	`enterVtbi`	YES	(`m.pump` = *this one* OR `m.interleave`) AND `inp.vtbi[pump]` AND `inp.dmode[pump] = dvtbi`
`chooseTime`	NONE	`confirmVtbi`	YES	(`m.pump` = *this one* OR `m.interleave`) AND (`NOT(inp.time[pump])` OR `NOT(m.timeSet[pump])`)
`enterTime`	`enterVtbi`	`chooseTime`	YES	(`m.pump` = *this one* OR `m.interleave`) AND `NOT(inp.time[pump])` AND `inp.dmode[pump] = dtime`
`confirmTime`	NONE	`enterTime`	YES	(`m.pump` = *this one* OR `m.interleave`) AND `inp.time[pump]` AND `inp.dmode[pump] = dtime`
`openClamp`	`enterTime`	`confirmTime`	NONE	(`m.pump` = *this one* OR `m.interleave`) AND (`inp.vtbi[pump]` OR `m.vtbiSet[pump]`) AND (`inp.time[pump]` OR `m.timeSet[pump]`)
`startInfusion`	(`openClamp` AND `m.pump` = *this one*) OR `startInfusion` *on the other pump*	`openClamp`	YES	`inp.dmode[pump] /= dinfusing` AND *for both pumps:* (`inp.vtbi[pump]` OR `m.vtbiSet[pump]`) AND (`inp.time[pump]` OR `m.timeSet[pump]`)

particular pump or has chosen to interleave programming), "`NOT(inp.time[pump])`" (the user perceives that the time value currently displayed is different from the prescription value), and "`inp.dmode[pump] = dtime`" (the user perceives that the pump is in the time entry mode). Table 1 specifies action cues for programming one pump. In the two pump scenario considered, the specifications for both pumps simply duplicate that given in the table. This model layer also has a boolean parameter, `costs`. It is true, when the costs of accessing information (prescription form) are assumed to be high, and false otherwise.

These assumptions focus on the distinction between the task-orientated and device-orientated steps [13]. The device-orientated steps are potentially more problematic because they have lower activation levels than their task-orientated counterparts. These lower activation levels are assumed to be the result of the different ways in which device- and task-orientated steps are represented in a mental model. It is assumed that device-orientated steps are associated only with the device-knowledge cues, while task-orientated steps are associated with both task- and device-knowledge cues. Table 1 shows that the actions `confirmVtbi` , `chooseTime` and `confirmTime` are assumed to be device-orientated; they do not have task-knowledge cues. All other actions are cued on the task-knowledge level. The action `memorise`, if taken, is the first in a series. Therefore, it is assumed to be cued by the task goal.

As can be seen in Table 1, it is also assumed that all actions that involve the buttons on the front panel of an infusion pump are sensorily cued. On the other hand, the roller clamp (positioned at the side of the pump) provides no sensory cues for the action `openClamp` . It is also assumed that there is no sensory cueing for the action `memorise` .

3.3 The System Model

The specification of the interactive system as a whole involves combining and connecting the device model and the user model. This requires two additional models: firstly that of user interpretation of the device interfaces (`Interpretation`) and the environment, and secondly a model giving the effect of user actions on the pumps (`Effect`). These additional models connect the state spaces of the device and user models. These connectors are in fact simple. The `Interpretation` model renames appropriate variables as in the following case:

```
inp.dmode = dmodes
```

For the values of the infusion parameters (e.g., VTBI) such renaming takes into account whether the relevant value is displayed by the pump:

```
inp.vtbi[pump] = (vtbis[pump] and vtbisDisp[pump])
```

Finally, the perception of a prescription form is assumed to depend on the costs of consulting it. If these costs are considered to be low, a prescription form can always be perceived (consulted):

```
inp.prescription[pump] = (costs = FALSE)
```

The effect of user actions is specified in `Effect` by stating that the input event on `pump` (`events[pump]`) is either a "do nothing step (`tick`) or whatever action (`out.action`) the user model produced.

The SAL module `System` of the interactive system is then specified as the following composition of all these separate models:

```
(User || Effect) [] (Pumps || Interpretation)
```

The structure of this composition also applies to other interactive systems involving different devices.

4 Verification-Based Analysis

Given the model as specified in the previous section, the aim is to explore potential usability problems that might arise through the use of this interactive device under the cognitive assumptions made. The impact of the costs of accessing information are of particular interest. The aim is to generate predictions about use that can be compared with the results of an experimental study.

SAL model checking tools were used to analyse the properties of the interactive system model. The cognitive assumptions about user behaviour (a 'surrogate' user) help to identify unforeseen interaction issues by asking general questions: for example, does the user model always achieve the task goal? In the example, such a question is formulated as the following LTL property `goal` (`F` means 'eventually'):

```
F (task)
```

The property states that, in any interactive system behaviour, the task goal is eventually achieved (i. e., user perception indicates that both pumps are infusing). However, starting infusion before a roller clamp is opened is *unsafe*. Thus the following LTL property, `safe`, is formulated to check if that holds (`G` means 'always):

```
G ((dmodes[1] = dinfusing => not(clamps[1])) and
   (dmodes[2] = dinfusing => not(clamps[2])))
```

This property checks, for each pump, whether its roller clamp is open whenever the pump is infusing.

The analysis starts with the assumption that the costs of consulting a prescription form are low (`costs` was set to `false` in `System`). Indeed, before programming a pump, it makes sense for a nurse to position a prescription form nearer to the device so as to minimize the cost of looking back and forth between each. In this case, model checking the property `safe` produces a trace that describes a roller clamp error on the first pump. It is not obvious how to change the design of the infusion pump itself to prevent this error of forgetting to open the roller clamp. However, this error can be explained by the interleaving behaviour in programming two pumps, while such behaviour may be encouraged by easy access to a prescription form. Therefore, it is plausible to hypothesise that increasing the costs of accessing prescription may help to avoid the omission error on the roller clamp step.

The next step in the analysis is to verify this hypothesis by setting `costs` to `true` in the model. In this case, model checking both properties `safe` and `goal` succeeds. The successful verification can be interpreted as a prediction that any design change in the interactive environment, that increases the costs of accessing the prescription values, helps to prevent the omission error. In practice, such a change can be achieved in several ways: for example, by positioning a prescription form further away from the pumps, or by chunking prescription values (VTBI and time) for each pump on the form. The impact of such changes on the safety of pump programming can be further explored in experimental studies.

5 The Experiment

As already discussed one of the aims of the experiment was to investigate how the design of an interactive environment impacts on safety when programming two infusion pumps simultaneously. The experiment demonstrates that the seemingly sensible action to position a prescription form nearer to the device increases the likelihood of the error of forgetting to open the roller clamp when programming two pumps.

Method. Back et al [14] describe an experiment that investigates how the physical layout of the environment impacts on participants' interleaving behaviour when programming two infusion pumps. Participants were invited to program the pumps using information from a prescription form. The physical and mental effort involved in accessing information was manipulated by varying the physical distance between the prescription form and the devices.

The soft constraints hypothesis [3] maintains that when selecting between low-level memorisable procedures, those that tend to minimise performance cost while achieving expected benefits will be selected. Performance cost can be measured in terms of time for example. Depending on the situation, a perceptual strategy (where the prescription form is consulted when the relevant value is needed) may be more efficient than a memory-intensive one of memorising both prescription values at once. In the low information access cost condition, the soft constraints hypothesis suggests that people are more likely to use a perceptual strategy, when retrieving information needed to program a device, over a memory-intensive one.

Results. Participants were only able to use a low-level strategy when the prescription form was located alongside the devices being programmed. Critically when adopting a perceptual strategy, value entry may be driven by prompts from the devices, rather than what values are held in memory. A user may continue to enter values, consulting the prescription form and interleaving between devices as necessary, until all requested values are entered. Experimental data showed that adopting a perceptual strategy encouraged interleaving during device programming, which resulted in an increased omission error rate. Such errors were rare when people chose not to interleave until they finished programming one device.

Generally, these results corresponds to the predictions based on the verification of the interactive system model. However, the specific example of erroneous behaviour generated by the model suggests a different interleaving behaviour than the ones observed in the experiment. This discrepancy could be a behaviour that could plausibly happen in reality but was just not seen in the (limited) experiment. It may alternatively suggest that the cognitive assumptions believed to apply (so modelled in the intermediate layer) in this situation are actually insufficient. In particular it may be that there is something more that matters than is captured by the specified distinction between task- and device-oriented interaction steps. Alternatively, it could be that the set of concrete modelling assumptions about action salience and cueing (Section 3.2) is not sufficiently precise. In either case, the modelling has drawn attention to something that needs further investigation if the experimental results and so the usability of the design are to be fully understood. On the other hand, systematic experimentation can be used to validate the generic user model [12].

6 Conclusion

This paper has described a novel approach to evaluating the usability of an interactive device design using a formal method that focuses on the experimentation associated with user evaluation. The technique helps the experimenter to interpret results formatively to improve a potential design. It also makes more general predictions (e.g., about the impact of the costs of accessing information) as opposed to specific conditions and scenarios that are investigated in experiments. In the example an abstract description of a design of the interactive system was produced that not only described the device but also provided information about other resources, such as prescription forms, that could be used in the interaction. The results of the evaluation indicate potential changes to the larger system — the context in which the interactive device is to be used.

It is typically difficult both to interpret the results of usability evaluation, and to make appropriate changes to the design of an interactive system as a result of the evaluation. The approach presented in this paper uses formal methods in a novel way, integrating it with laboratory studies to improve an iterative design process in relation to the development of interactive systems.

There are several issues that require further study. Firstly, if these techniques are to be used effectively in bridging the gap between a formal specification of the interactive system design and its empirical evaluation then the assumptions that are captured by the interaction model must be developed in a format that is comprehensible and disputable by the evaluator who will come from a human factors tradition.

Secondly, the behaviours that are being investigated using this approach are intended to be error behaviours. These are behaviours that may be business or safety critical. Typically (and hopefully) these behaviours are rare. Experiment cannot always provide access to such errors and therefore other techniques intended to increase their likelihood must be chosen, for example by using secondary tasks. Aspects of experiments such as these are not explored using the modelling approach described in the paper.

These two issues are the basis for further study.

Acknowledgments. This project was partly funded by the CHI+MED project: Multi-disciplinary Computer Human Interaction Research for the design and safe use of interactive medical devices (UK EPSRC Grant EP/G059063/1). We are grateful to Jonathan Back and Anna Cox of University College London Interaction Centre for input relating to the experiment and cognitive assumptions.

References

1. Vicente, K., Kada-Bekhaled, K., Hillel, G., Cassano, A., Orser, B.: Programming errors contribute to death from patient-controlled analgesia: case report and estimate of probability. Canadian Journal of Anesthesia / Journal canadien d'anesthésie 50, 328–332 (2003)
2. Ritter, F.E., Young, R.M.: Embodied models as simulated users: introduction to this special issue on using cognitive models to improve interface design. International Journal of Human-Computer Studies 55, 1–14 (2001)
3. Gray, W.D., Sims, C.R., Fu, W.T., Schoelles, M.J.: The soft constraints hypothesis: A rational analysis approach to resource allocation for interactive behavior. Psychological Review 113(3), 461–482 (2006)
4. Bolton, M.L., Bass, E.J., Siminiceanu, R.I.: Generating phenotypical erroneous human behavior to evaluate human–automation interaction using model checking. International Journal of Human-Computer Studies 70(11), 888–906 (2012)

5. de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., Tiwari, A.: SAL 2. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 496–500. Springer, Heidelberg (2004)
6. Fields, R.E.: Analysis of erroneous actions in the design of critical systems. Technical Report YCST 20001/09, University of York, Department of Computer Science, D.Phil Thesis (2001)
7. Beckert, B., Beuster, G.: A method for formalizing, analyzing, and verifying secure user interfaces. In: Liu, Z., Kleinberg, R.D. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 55–73. Springer, Heidelberg (2006)
8. Bowman, H., Faconti, G.: Analysing cognitive behaviour using LOTOS and Mexitl. Formal Aspects of Computing 11, 132–159 (1999)
9. Barnard, P.J., May, J.: Interactions with advanced graphical interfaces and the deployment of latent human knowledge. In: Interactive Systems: Design, Specification, and Verification (DSV-IS 1995), pp. 15–49. Springer (1995)
10. Rushby, J.: Analyzing cockpit interfaces using formal methods. Electronic Notes in Theoretical Computer Science 43 (2001)
11. Altmann, E.M., Trafton, J.: Memory for goals: an activation-based model. Cognitive Science 26(1), 39–83 (2002)
12. Rukšėnas, R., Back, J., Curzon, P., Blandford, A.: Verification-guided modelling of salience and cognitive load. Formal Aspects of Computing 21, 541–569 (2009)
13. Ament, M.: The role of goal relevance in the occurrence of systematic slip errors in routine procedural tasks. Technical report, UCL, PhD thesis (2011)
14. Back, J., Cox, A., Brumby, D.: Choosing to interleave: human error and information access cost. In: Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems, CHI 2012, pp. 1651–1654. ACM, New York (2012)

Automatic Inference of Erlang Module Behaviour

Ramsay Taylor, Kirill Bogdanov, and John Derrick

Department of Computer Science, The University of Sheffield

Abstract. Previous work has shown the benefits of using grammar inference techniques to infer models of software behaviour for systems whose specifications are not available. However, this inference has required considerable direction from an expert user who needs to have familiarity with the system's operation, and must be actively involved in the inference process. This paper presents an approach that can be applied automatically to infer a model of the behaviour of Erlang modules with respect to their presented interface. It integrates the automated learning system StateChum with the automated refactoring tool Wrangler to allow both interface discovery and behaviour inference to proceed without human involvement.

1 Introduction

This paper presents an automated technique to reverse engineer state machine models of the behaviour of Erlang modules. Reverse engineering specifications in the form of Finite State Machines can be of considerable benefit to projects where the original specification has been lost, never existed, or no longer reflects the state of the system due to requirements changes. Additionally, where a system is to be updated or replaced, an understanding of the behaviour of the current system behaviour is required to form a regression test for the replacement.

Erlang (described in more detail in Section 2.1) is a language designed to support communicating concurrent systems. As such, Erlang software can consist of many independent modules that implement particular processes within the overall system. Erlang was originally intended for use in telecoms infrastructure, so it contains features to support the in-place replacement of live, running modules. As such, gaining a proper understanding of an Erlang module can be vital if a replacement is to be made without damaging the integrity of the overall system.

Grammar inference (or *language learning*) algorithms, which are described in more detail in Section 2.3, have been shown to be useful in determining the behaviour of software systems [6,15,17,14]. However, previous work on inferring behaviour from Erlang systems [14,16] required considerable human involvement in the abstraction of the data into traces, and in answering the queries generated by the inference algorithm. Some effort has been made to automate the process [17] but this still requires human input in instrumenting the system under test, and in selecting and presenting the initial traces to the learning system. Additionally, since the learning algorithms that are considered here require both *positive* and *negative* traces there has to be some definition of failure encoded by a human user.

The principle contribution of this paper is to remove the human involvement as far as possible from the incremental stages of system behaviour inference. The structure of Erlang [5] makes this automation practical, as Erlang modules can implement defined *behaviours* that place specific requirements on their interface. This defined interface, combined with Erlang's language features for executing function calls in monitored child threads makes the automatic execution and evaluation of program traces practical.

The process is presented as three phases:

– **Alphabet determination** The automatic inference of the interface of the System Under Test.
– **Automated query evaluation** The automatic evaluation of proposed program traces, their classification as positive or negative, and the collection of sample output.
– **Active behaviour inference** A query-based process of learning that generates an adequate test set to evaluate the system's behaviour.

The inference process utilises the StateChum [2] learner, which provides a framework for grammar inference implemented in Java. This was augmented with an Erlang wrapper layer (described in detail in Section 4) to facilitate trace execution and evaluation. The instrumentation that is undertaken in Section 3 makes use of the API for the Wrangler [11] Erlang refactoring system to instrument available test cases, or the TypEr [12] type inference system to infer types and interfaces. The workflow outline is presented in Figure 1.

Together, these components are composed into an entirely automated workflow that, when applied to an Erlang module, can perform interface discovery, test generation, and active learning with feedback from dynamic tests, without any human involvement. The prototype software used to generate the results presented in this paper is available from the author's website[1].

The remainder of this paper is structured as follows: Section 2 contains descriptions of Erlang, the OTP framework, and the learning algorithms used in this work; Section 3 describes the process of inferring a suitable interface to the module; Section 4 describes the Erlang *Oracle* that can evaluate potential traces; Section 5 explains the active learning process itself; Section 6 contains conclusions.

2 Background

2.1 Erlang

Erlang [5] is a programming language originally developed at Ericsson for use in their telecoms infrastructure products. It is now available as open source

[1] `http://staffwww.dcs.shef.ac.uk/people/R.Taylor/ErlangInference/`

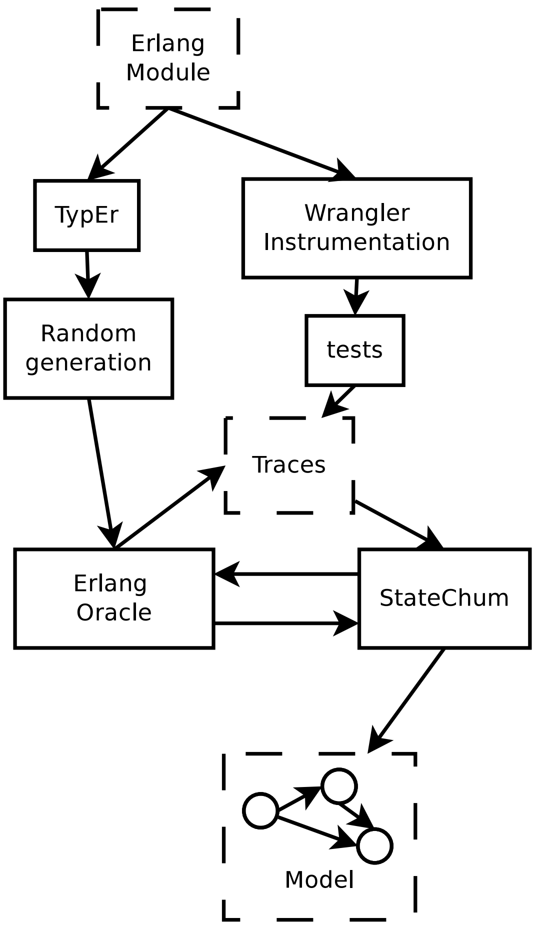


Fig. 1. The automated workflow

software. It is a declarative language but uses several components of the functional programming paradigm, such as pattern matching and extensive use of recursion.

The examples in this paper will use the Erlang *locker* module shown in Figure 2. This implements a single item storage space that can only be written to when in the locked state.

An Erlang module contains a number of functions, each of which is defined by a series of patterns starting with a name, a set of parameters, the arrow symbol `->`, the function definition, and ends with a full stop. Separate patterns are separated by semicolons and the final pattern terminated with a full stop. Patterns are matched in order with the first matching pattern being applied. Variable names begin with capital letters or the underscore character (e.g. `Var, _S`), whilst lower

```
 -module(locker).

-behaviour(gen_server).
-export([init/1, handle_call/2, handle_call/3, handle_cast/2, terminate/2]).

init(_) -> {ok, {unlocked, -1}}.

handle_call(lock, {unlocked, S}) -> {reply, {ok, locked}, {locked, S}};
handle_call(lock, {locked, _S}) -> erlang:error("Locked lock!");
handle_call(unlock, {unlocked, _S}) -> erlang:error("Unlocked unlock!");
handle_call(unlock, {locked, S}) -> {reply, {ok, unlocked}, {unlocked, S}};
handle_call(read, {State, S}) -> {reply, S, {State, S}};
handle_call({write, Val}, {locked, _S}) ->
   {reply, {ok, Val}, {locked, Val}};
handle_call({write, _Val}, {unlocked, _S}) ->
   erlang:error("Unlocked write!").

handle_call(Msg, _From, State) -> handle_call(Msg, State).

terminate(_Reason, _State) -> ok.
```

Fig. 2. The locker module — locker.erl

case letters indicate an "atom" value (conceptually a user defined keyword, e.g. `lock, unlock`). Tuples are contained in curly brackets ({`lock, S`}), lists in square brackets (`[a,b,c]`). Strings are treated as lists but can be presented in double quotation marks.

The `locker` server begins in the unlocked state, storing the value `-1` in its internal state. It will respond to the `call` operation, responding according to a number of different parameters. The atom `read` can be sent at any time to query the currently stored value. `lock` transitions to the locked state, and `unlock` performs the reverse. The tuple {`write, Val`} will replace the current value with `Val`, but will cause the system to exit with an exception if sent whilst the system is in the unlocked state. Sending `lock` whilst already locked, or `unlock` whilst unlocked also causes an error. All other parameter values are ignored.

Erlang is an interpreted language and so the failure of the interpreter to find a matching pattern for a particular function application is reported at runtime. There is an exception throwing model for error handling, which allows pattern matching over the types of exceptions caught. Erlang also features a process-oriented distributed programming model that uses asynchronous communication channels.

2.2 The Open Telecoms Platform (OTP)

The Open Telecoms Platform (OTP) describes an open source software collection released by Ericsson that includes the Erlang compiler and interpreter, along with various tools for Erlang analysis (including dialyzer and TypEr [12]) and a large collection of libraries to support the creation of Erlang systems using standard patterns. As these patterns are widely used, the inference technique presented in this paper uses them as a starting point for the automatic inference of the interface of the system under test.

The `gen_server` "behaviour", for example, describes a generic server system that accepts synchronous and asynchronous requests and responds, whilst maintaining some internal state that influences the responses. To implement a `gen_server` the programmer simply needs to provide a module containing "callback" functions. The OTP `gen_server` behaviour provides all of the mechanics of maintaining the Erlang processes and handling both the synchronous and asynchronous messaging. When the server process receives a request it calls the appropriate callback function with the request content and the current server state as arguments. The callback function determines what response is provided, and how the state of the server process evolves. The `locker` example is implemented as a `gen_server`, so it responds to the `call` and `cast` operations, the behaviour being defined by the `handle_call` and `handle_cast` functions that can be seen in Figure 2.

The OTP libraries do not interact with the state variable of the server process, so any form can be used. The locker example stores a pair of an atom and a user value in its state. The `lock` operation (the first pattern in `handle_call`) is defined for a state that is a pair containing the atom `unlocked`, and the stored value, which is bound to variable `S`. When the system is unlocked (as defined by the second pattern) it will move to the locked state, retaining its stored content, and replies with the message {`ok, locked`}. If the system is already locked then an exception is raised. This failure mode will be identifiable by the automatic learning described in the remainder of this paper.

In the `gen_server` behaviour the `cast` operation is performed asynchronously — that is, the calling process continues execution immediately — whereas the `call` operation blocks the calling process until a response is received. The learning process utilised in this paper is a FSM learner and so expects responses to be synchronised to calls. Consequently, asynchronous communications are not considered, however it might be possible to utilise some ideas from [7] to cover some of these cases.

2.3 Language Learning Algorithms for Software Test Generation

Algorithmic approaches to language learning started in the 1960s with Gold [9], building on earlier work to formalise natural language. In the 1980s, Anlguin published the L* algorithm [4] that learns a language from an expert oracle by presenting queries. The queries take the form of sequences that may or may not be in the language, and the expert oracle replies simply whether they are or are not valid language elements. The L* algorithm queries the language incrementally and exhaustively, attempting each alphabet element from each state and iterating until no new states are identified. This produces the complete automaton but requires lengthy exploration of the language and heavy use of an expert oracle.

Later work has aimed to learn state machine language representations from partial data, usually a subset of the possible sequences of the language. Evidence driven state merging (EDSM) algorithms such as BlueFringe [10] operate on a set of positive and negative traces from a system or language and produce a state machine that accepts positive traces and rejects (in the form of a transition to a failure state) negative traces. The algorithm was shown to be highly effective in the "Abadingo One" competition for learning algorithms, but does not require the use of an expert oracle, operating exclusively on the presented positive and negative traces. The QSM algorithm [8] reintroduces the oracle query approach by producing queries from the new traces that are possible in the system after the proposed merge and requires the oracle to either confirm that they are positive traces, or supply the shortest negative prefix. If the response is the same as the classification of those traces by the outcome of the proposed merge then QSM will continue to suggest merges. If the response contradicts the merged automaton the algorithm restarts from the beginning, with this additional negative trace. Eventually enough information is accumulated for the learning process to converge.

Non-exhaustive language learning algorithms have significant application to software testing as they present an opportunity to explore the behaviour of the software system, without the resource and time requirements of exhaustive model checking. The StateChum system was developed to implement the QSM algorithm with the objective of reverse engineering state machine representations of software behaviour from software trace data [16].

To learn the behaviour of the `locker` example presented in Figure 2 it is first necessary to produce a set of positive and negative traces. A system expert, who already has a reasonably understanding of the operation of the system, must first identify the interface elements that will form the alphabet of the traces. Positive traces can often be extracted from log data, but negative traces must be prepared by the system expert who can identify usage patterns that are not accepted.

Initially, known traces are represented as a tree by merging common prefixes. A possible tree for the `locker` example is shown in Figure 3.

The manual process of inference using QSM begins by presenting the user with a query in the form of a trace, such as {`call,init`} {`call,unlock`}. The query requires that the user either accepts the trace as valid in the system, or identifies the first point at which it is rejected. It is also possible for formulae to be entered in order to answer questions [14].

The response to this query is used to update the system model and a new query is presented. This continues until no more states can be merged. The number of queries that must be processed manually can be considerable, even for a small system such as this.

The work presented in the remainder of this paper uses the same core learning system from StateChum but automates the process of answering queries. For the same initial trace set the automated system presented here was able to infer the correct state machine (presented in Figure 8) in 5.8 seconds with no human input on a single core of a 2GHz Intel Xeon.

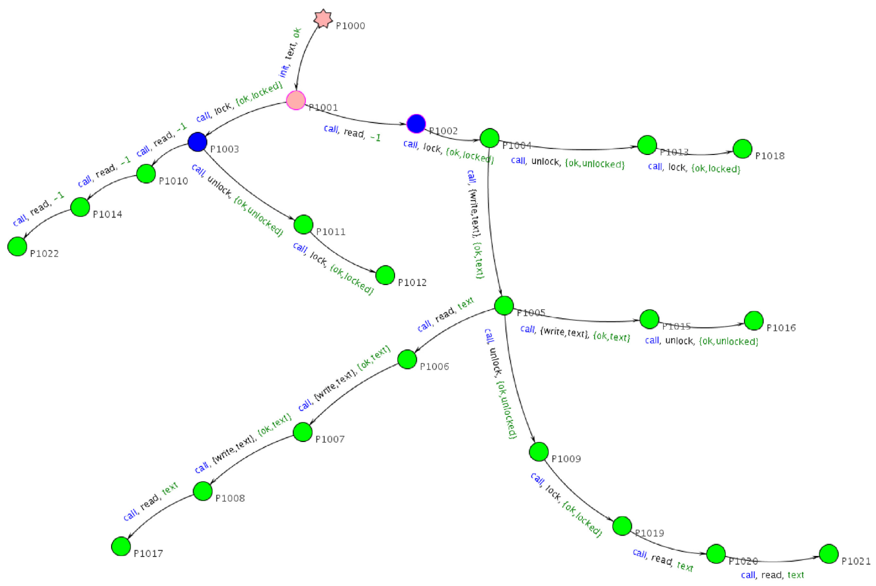


Fig. 3. The Prefix Tree Automaton (PTA) produced before QSM queries begin

3 Alphabet Determination

This section presents the integration of the refactoring tool Wrangler [11] with the newly developed Erlang tracing components (described in Section 3) to fully automate the process of interface discovery for Erlang modules.

To begin learning the behaviour of a module it is first necessary to determine its *alphabet*, that is the externally observable and accessible actions that can be performed. Erlang modules specify a list of *exported* functions, and this is the most general alphabet of any module. However, some Erlang OTP libraries require that various handlers and callback functions be exported to allow the library modules to use them where necessary. This can result in Erlang modules having a mixture of genuine interface functions and internal callback functions in their export list.

Where a module implements one of the OTP behaviours it is expected that the module will be accessed through the relevant behaviour library functions. This presents a more consistent and predictable interface. Using the behaviour to inform the choice of interface has the advantages that there will be a known set of relevant functions, and the standard functions only take one argument (although this can be of arbitrary complexity).

Having identified the functions that form the module's interface, it is also necessary to determine relevant values for the parameters. This work is directed at inferring FSM models of software, and not *Extended Finite State Machine*

(EFSM) models, and so parameter values are condensed into the transition labels (e.g. $f(1)$ and $f(2)$ are considered to be different and independent events). However, some parameter values are necessary to seed the learning process. In the case of the `locker` example the simple pattern matching nature of the handler functions makes it easy to determine a suitable set of parameters, but Erlang functions are often written with more subtle pattern matching and with more complex internal behaviour. Additional information can be provided by the TypEr [12] type inference system. Erlang is not an explicitly typed language like Haskell, so the TypEr system was designed to infer the type signature of Erlang functions. When run on the `locker` module TypEr infers that the available argument values for `handle_call` are `lock`, `read`, `unlock`, and {`write, Val`}.

When combined with the known structure of the OTP behaviour handler requirements this can be interpreted automatically to derive the range of possible patterns that this function responds to. Although irrelevant in this example, TypEr performs inference on the entire function definition, not only the pattern headers. If the function definition contains unconstrained variables but the defined behaviour identifies particular types using Erlang's pattern-sensitive choice operations, then this information will be reflected in the type signature.

These type signatures may not define the entire behaviour of the module but are a useful basis for an active learning system such as QSM to explore. As well as being more general than simple text analysis, TypEr has the advantage that it can be run on compiled `beam` files — so long as they were compiled with the `debug_info` flag — without access to the source code. The critical disadvantage of the TypEr inference system is its production of the most general type signature for the function. Well written Erlang function definitions utilise defensive coding techniques that include "catch all" patterns to gracefully handle unexpected inputs. This produces more robust software but it results in TypEr inferring the most general type `any()` for all such functions, which gives no guidance as to useful parameter values.

In the case that the source code is available but a function's type is too general it is necessary to gather some example usage. To produce some sample values the code can be instrumented and inserted into a live system. Alternatively, if a test set exists, this can be run on the instrumented code. For this work the instrumentation was provided automatically using the Wrangler refactoring system[11]. Wrangler is a refactoring system that presents an emacs interface, but it also contains a programatically accessible API. The Wrangler API allows refactorings to be specified in an elegant template format. Using this system it was possible to automatically insert logging calls into each of the functions that were relevant to a module's alphabet. This produced a log that consisted of a sequence of functions names and parameter values. The code shown in Figure 4 demonstrates how the Wrangler API makes this a simple procedure.

On its own, this provides suitable information to derive some sample parameter values, but by instrumenting the test functions it was possible to break the log file into sections corresponding to distinct tests. These form traces of the

```
specific_log_mutation(File, FunctionNames, LogID) ->
  ?FULL_TD_TP([?RULE(?T("f@(Args@@) when Guard@@-> Body@@;"),
    begin
      {NewArgs@@, LogArgs@@, _} = convert_args(Args@@, 1),
      ?TO_AST("f@(NewArgs@@) when Guard@@->
              mu2_logger:log(?MODULE, f@, [LogArgs@@], "
              ++ LogID ++ "), Body@@;")
    end,
    contains(FunctionNames, {list_to_atom(?PP(f@)), length(Args@@)}))],
    [File]).
```

Fig. 4. The Wrangler API code to apply function instrumentation

system, but they do not contain the results of the function calls, nor do they encode any sense of failure. The next section resolves this problem.

4 Query Evaluation

This section describes the development of an Erlang wrapper system to dynamically execute and observe traces of a System Under Test. This provides the classification of artificially generated traces, additional information as to the response alphabet of the functions, and the ability to automatically answer the queries generated by the active learning in Section 5.

Having determined a suitable alphabet for the behaviour inference it is possible to propose some possible traces of the system by simple, random concatenation of alphabet elements that can then be evaluated. Alternatively, the instrumentation described in Section 3 can provide some sample traces, but these will be limited by the available stimulation for the instrumented code.

In either scenario there will be areas of the module's behaviour that have not been explored. In Section 5 the process of expanding the covered behaviour is discussed, but it will require an *oracle* that can execute potential traces and classify them as positive or negative.

The traces must be encoded in a way that is usable by standard learning tools such as StateChum, which generally expect traces to be composed of distinguishable strings, but also allow the oracle to evaluate the behaviour of queries. To facilitate the evaluation the alphabet elements are encoded as Erlang tuples containing the name of the function to call (or the behaviour library action), the parameters to use, and (optionally) the expected result, e.g.:

```
{init,[]} {call,read,{ok,-1}}
```

The first tuple encodes the use of the `init` action in the `gen_server` behaviour library, with the empty list as parameters and it places no requirements on the result. The second tuple encodes the use of the `call` action with the parameter `read` and expects the result {`ok,-1`}.

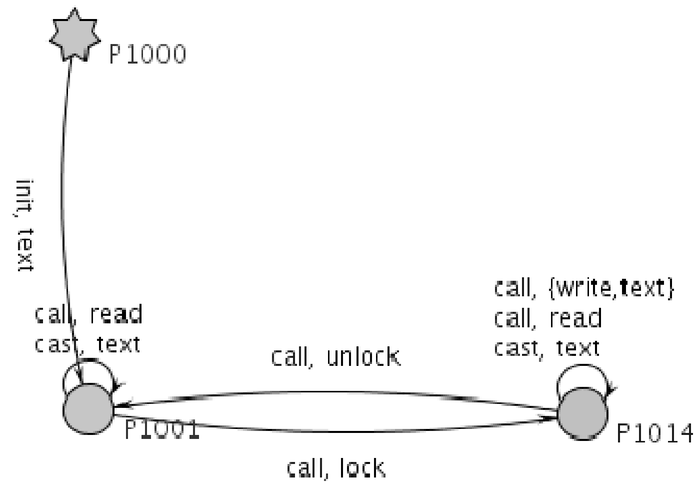


Fig. 5. Incomplete locker state machine

It is possible to infer some behaviour by simply observing the sequence of function calls performed and considering those that cause the system to crash or throw an unhandled exception as failures. In the locker example this will identify that `init` must be called first, and that `write` cannot be performed in the unlocked state, as shown in Figure 5. However, this misses an important detail of the `write` operation: that it changes the internal state of the system such that `read` operations will return the newly-written value instead of the old one. To capture this detail it is necessary to record the output of the system in response to specific operations. Erlang systems can have multiple outputs in various forms but this work limits its consideration of output to just the function return value. In the case of `gen_server` operations — such as `call` — this makes sense since they have an explicit mechanism to return a value to the calling process.

The Erlang oracle can evaluate a trace (in the form of a sequence of these tuples) by executing each action with the specified parameters and comparing the received output to the expected output. Since StateChum operates on prefix closed systems the evaluation can terminate immediately if either the system is observed to crash after a particular sequence, or if the received output is different from the expected output. In the latter case the oracle will report the trace up to that point as negative (since the required output cannot be observed) but it will also report a positive trace with the actually received output in the last tuple. In this way the learner will gain an additional transition, and possibly an additional alphabet element if the response has not been previously recorded, for example:

```
- {call,init} {call,lock} {call,{write,text},{ok,text}}
```

```
                                        {call,read,{ok,-1}}
+ {call,init} {call,lock} {call,{write,text},{ok,text}}
                                      {call,read,{ok,text}}
```

A pseudocode algorithm representation of the trace component is shown in Figure 6.

```
procedure TRACER($MUT, Wrapper, Trace$)
    $Pid \leftarrow$ spawn $Wrapper(MUT, Trace, self())$
    $Observed \leftarrow []$
    repeat
        $Elem \leftarrow$ receive from $Pid$
        $Observed \leftarrow Observed ++ Elem$
    until $(Observed = Trace) \vee (Pid$ dies$)$
    if $Observed = Trace$ then return $\{accept, Trace\}$
    else
        $Elem \leftarrow nth(length(Observed) + 1, Trace)$
        return $\{reject, Observed ++ Elem\}$
    end if
end procedure
```

Fig. 6. The tracer algorithm

This evaluation can respond to the queries generated by the active learning process described in Section 5, but it can also be used to classify randomly generated traces, or to fill in details such as uncaptured output data in instrumentation logs. By classifying traces from these sources it is possible fully automatically to build a suitable set of traces for the active learning process.

5 Behaviour Inference

The previous sections have described automated processes to determine the alphabet of an Erlang module, and to evaluate traces over the module. By utilising the Erlang oracle to answer QSM queries with the StateChum inference system the process for inferring a state machine model can be completely automated.

To allow automatic learning an initial set of system traces is required. If instrumentation of a running system has been possible then this may provide some positive traces, but some negative traces are also required. To generate the initial traces to seed StateChum's QSM algorithm it is adequate to produce random sequences of alphabet elements and present them to the Erlang oracle for classification. The oracle will either verify them as positive traces, or produce the negative prefix. The selection of suitable initial traces will partly depend on the behaviour of the module itself. A balance between breadth and depth must be found. Some behaviour will not become apparent until a certain trace depth, such as the write behaviour of the `locker` example, which cannot be detected without at least the positive traces:

```
{call,init} {call,read,{ok,-1}}
{call,init} {call,lock} {call,read,{ok,-1}}
{call,init} {call,lock} {call,{write,text},{ok,text}}
                                   {call,read,{ok,text}}
```

Pure depth is also not adequate, since such traces do not highlight the variety of the behaviour of the `lock` event, because `write` is not attempted prior to the `lock`. The exact combination of breath and depth will vary between modules so the StateChum implementation leaves these choices to the user.

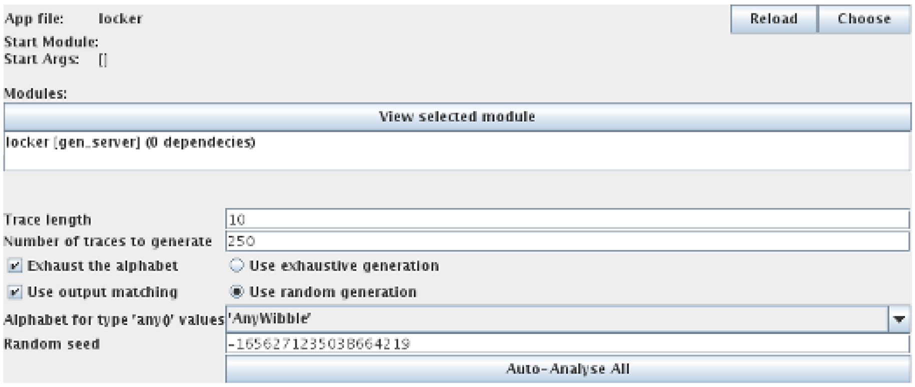

App file: locker
Start Module:
Start Args: []
Modules:
locker [gen_server] (0 dependecies)
Trace length: 10
Number of traces to generate: 250
[✓] Exhaust the alphabet
[✓] Use output matching
Use exhaustive generation [✓ Use random generation]
Alphabet for type 'any()' values: 'AnyWibble'
Random seed: -1656271235038664219

Fig. 7. StateChum trace generation interface

The seed interface shown in Figure 7 provides control of the generation process and allows the reuse of a random seed value for repeatability. The exhaustive generation method is included for comparative value but it is impractical for alphabet sizes and trace depths that are larger than trivial values. The "exhaust alphabet" option will attempt every element of the alphabet as a single element trace. In many cases most of these will fail since they are not an `init` element, but having the elements present in the trace file informs QSM of their existence, which then allows it to attempt those transitions at other points in the state machine.

Finally, StateChum was modified to present queries via the Ericsson Java-Erlang bridge [1] to the Erlang oracle. This allows hundreds of queries to be answered per second, rather than the labourious process of manual responses. Also, it does not depend on an expert user, as the answers are provided by observation of the implementation.

A complete workflow that produces a state machine model of a module's behaviour automatically is presented in Figure 1.

The final FSM inferred by this process appears in Figure 8. Clearly, subsequent writes with different values will produce even more states. It is also possible for this system to correctly unify the result of changing the system's internal

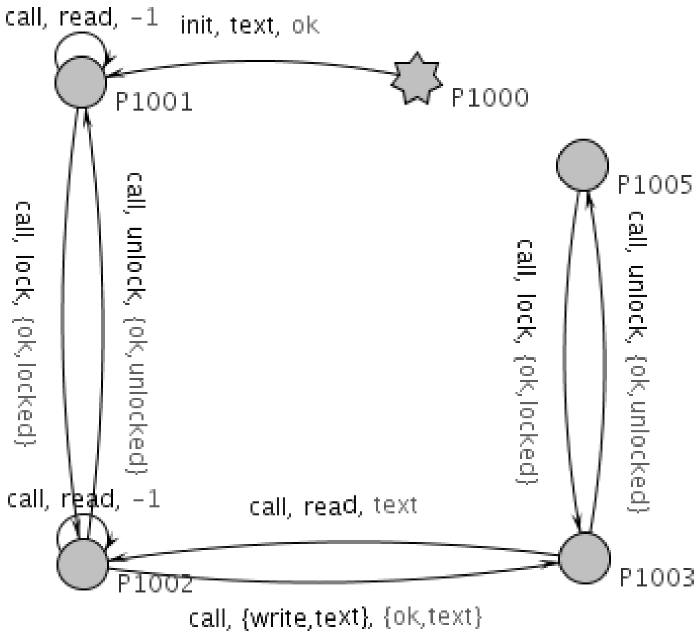


Fig. 8. Expanded locker state machine

variable back to a previous value with the state of the system at that previous point. Since there are infinitely many possible internal values this is no longer a regular grammar, and it is impossible for QSM to completely infer the possible behaviour of the system. The limit of exploration will depend on the choices of input parameters when the alphabet was created in Section 3. With output matching a trace length of 7 must be supplied to the random trace generator, and then learning is complete in approx. 5 seconds.

6 Conclusions

This paper has presented the application of several existing and newly developed automated techniques to the inference of Erlang system behaviour. The elements are combined into the workflow shown in Figure 1 to produce an entirely automated process that can be presented with an Erlang module and can produce a state machine model of the module's behaviour.

A significant limitation of this technique, which is common to all active learning techniques, is that some functions require parameters that can not be synthesised for trace execution — for example, the process IDs of other system components, which are regenerated every time the system is re-initialised. Such parameters can be captured by the instrumentation described in Section 3 and can be used by passive learning (which StateChum also implements), but they frustrate the generation and evaluation of queries for an active learner. Some attempts have been made to avoid this problem [13], which operate by hiding un-sythesisable parameters if possible, and similar approaches could be applied to Erlang modules.

The integration of the separate automatic systems (StateChum and Wrangler) presented here demonstrates that many of the difficulties in the application of learning algorithms — such as the large number of queries and the requirement for an oracle — can be mitigated by combination with techniques and tools from other areas of program analysis and verification. Future work will aim to identify and apply other techniques to the problems that remain, for example applying machine learning approaches [3] to mitigate difficulties with parameter values.

References

1. jinterface, `http://www.erlang.org/doc/apps/jinterface/jinterface_users_guide.html` (accessed January 25, 2013)
2. StateChum, `http://statechum.sourceforge.net/` (accessed January 14, 2013)
3. Weka 3 - Data Mining with Open Source Machine Learning Software in Java, `http://www.cs.waikato.ac.nz/ml/weka/` (accessed January 27, 2013)
4. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. 75, 87–106 (1987)
5. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in ERLANG. Prentice-Hall (1996)
6. Arts, T., Seijas, P.L., Thompson, S.: Extracting quickcheck specifications from eunit test cases. In: Proceedings of the 10th ACM SIGPLAN workshop on Erlang, pp. 62–71. ACM (2011)
7. Bogdanov, K.: Test generation for X-machines with non-terminal states and priorities of operations. In: Fourth IEEE International Conference on Software Testing, Verification and validation, ICST (2011)
8. Dupont, P., Lambeau, B., Damas, C., Van Lamsweerde, A.: The QSM algorithm and its application to software behavior model induction. Applied Artificial Intelligence 22, 77–115 (2008)
9. Gold, E.M.: Language identification in the limit. Information and Control 10(5), 447–474 (1967)
10. Lang, K.J., Pearlmutter, B.A., Price, R.A.: Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In: Honavar, V.G., Slutzki, G. (eds.) ICGI 1998. LNCS (LNAI), vol. 1433, pp. 1–12. Springer, Heidelberg (1998)
11. Li, H., Thompson, S.: A User-extensible Refactoring Tool for Erlang Programs. Technical report, University of Kent (2011)
12. Lindahl, T., Sagonas, K.: Typer: a type annotator of erlang code. In: Proceedings of the 2005 ACM SIGPLAN workshop on Erlang, ERLANG 2005, pp. 17–25. ACM (2005)
13. Vaandrager, F.: Active Learning of Extended Finite State Machines. In: Nielsen, B., Weise, C. (eds.) ICTSS 2012. LNCS, vol. 7641, pp. 5–7. Springer, Heidelberg (2012)

14. Walkinshaw, N., Bogdanov, K.: Inferring finite-state models with temporal constraints. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE 2008, pp. 248–257. IEEE Computer Society Press (2008)
15. Walkinshaw, N., Bogdanov, K., Damas, C., Lambeau, B., Dupont, P.: A framework for the competitive evaluation of model inference techniques. In: Proceedings of the International Workshop on Model Inference in Testing, MIIT (2010)
16. Walkinshaw, N., Bogdanov, K., Holcombe, M., Salahuddin, S.: Reverse engineering state machines by interactive grammar inference. In: Proceedings of the 14th Working Conference on Reverse Engineering (WCRE), IEEE (2007)
17. Walkinshaw, N., Derrick, J., Guo, Q.: Iterative refinement of reverse-engineered models by model-based testing. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 305–320. Springer, Heidelberg (2009)

Integrating Proved State-Based Models for Constructing Correct Distributed Algorithms

Manamiary Bruno Andriamiarina[1], Dominique Méry[1], and Neeraj Kumar Singh[2]

Université de Lorraine, LORIA, BP 239, 54506 Vandœuvre-lès-Nancy, France
{Manamiary.Andriamiarina,Dominique.Mery}@loria.fr
Department of Computer Science, University of York, United Kingdom
neeraj.singh@cs.york.ac.uk, Neerajkumar.Singh@loria.fr

Abstract. The verification of distributed algorithms is a challenge for formal techniques supported by tools, such as model checkers and proof assistants. The difficulties lie in the derivation of proofs of required properties, such as safety and eventuality, for distributed algorithms. In this paper, we present a methodology based on the general concept of refinement that is used for developing distributed algorithms satisfying a given list of safety and liveness properties. The methodology is a recipe for reusing the old ingredients of the classical temporal approaches, which are illustrated through standard example of routing protocols. More precisely, we show how the state-based models can be developed for specific problems and how they can be simply reused by controlling the composition of state-based models through the refinement relationship. The *service-as-event* paradigm is introduced for helping users to describe algorithms as a composition of simple services and/or to decompose them into simple steps. Consequently, we obtain a framework to derive new distributed algorithms by developing existing distributed algorithms using *correct-by-construction* approach. The *correct-by-construction* approach ensures the correctness of developed distributed algorithms.

Keywords: Distributed algorithms, state-based models, composition, correct-by-construction, Event-B, liveness, eventuality.

1 Introduction

The formal modelling of distributed algorithms constitutes a challenge for methods and tools: these algorithms can be used to evaluate strengths and weaknesses of formal techniques supported by tools, such as model-checkers [12] and proof assistants [23, 27]. Formal techniques address properties, like safety, liveness and fairness. However, formal design and study of distributed algorithms also introduce other constraints to take into account, including time aspects [25], probabilistic features [17], fault-tolerance, scalability, dependability, etc. The *correct-by-construction* paradigm [15] offers an alternative and a promising approach to prove and derive *correct* distributed algorithms using a progressive and validated methodological approach [7]. More precisely, *refinement* is a key concept for organizing or structuring the (re-)development and (re-)discovery of distributed algorithms [2, 20] by reusing or replaying the former developments.

In this paper, we present a way to organise incremental refinement-based designs of distributed algorithms. Our methodology is based on structures coping with the modelling of distributed algorithms and providing a semantical framework for expressing both safety and liveness properties. We provide a list of recipes for reusing the old ingredients of the classical temporal approaches, by integrating refinement. Refinement-based development necessitates guidelines for helping the user to develop systems; these guidelines have to be able to incorporate refinement and make the composition of different interacting systems as simple as possible. When dealing with composition of interacting systems, there are more elements to prove, since we should demonstrate that the interacting systems are without interference [21]. We propose to minimize the complexity of proofs and formalisation, by *reusing* previous developments and proofs, and by *organizing* them. We introduce here a *component-driven* development of an algorithm: the *service-as-event* paradigm. A *component* actually represents a *phase* or a *step* of the algorithm: for instance, there are *initialisation*, *requesting critical sections* or *stabilisation* phases. It should be noted that we work on distributed algorithms exhibiting the following properties: the algorithms can be divided into components that describe phases local to nodes; and these phases are coordinated and synchronized according to the local states of the nodes. The main goal is to reuse as much as possible previous proofs of former refinement-based developments to model the phases.

The current approach extends the methodology described in [19] for developing *sequential* programs and *combining* phases, that we have experienced on algorithmic classical case studies related to the distributed protocols. We have noticed that *graphical (sequence)* diagrams can be used for expressing possible scenarios or phases of the protocols/algorithms, as defined by Tanembaum in [26]. The initial objective was to integrate such *graphical* notations, for *co-proving* a sequential program characterised by a *pre/post* specification. In a large number of cases, by using these diagrams and applying the *correct-by-construction* paradigm, we are able to derive algorithmic solutions annotated with invariants that can be checked and verified. However, these diagrams are not abstract enough to express properties on traces and fairness, and are also difficult to refine, while preserving properties such as fairness or liveness. Therefore, we limit the usage of these diagrams to the identification of algorithmic phases. The phases of an algorithm are defined by an *initial* (PRE) and a *final* (POST) states; and these phases are linked *sequentially*, by *temporal* operators, like *leads to* ($\leadsto$). The purpose of our work is to *link* and *coordinate* phases to obtain targeted distributed algorithms, by *integrating* and *composing* formal models, using *refinement diagrams* and the *service-as-event* paradigm.

Our paper is organised as follows. Section 2 introduces the modelling framework. Section 3 depicts the temporal framework for refinement-based development, more precisely state properties and refinement diagrams. Section 4 discusses structures for refinement-based development: the temporal coordination and decomposition of models, using the *service-as-event* paradigm. Section 5 illustrates our methodology with the study of the protocol ANYCAST RP. Finally, Section 6 concludes this paper along with the future work.

2 Choice of a State-Based Modelling Language

We choose EVENT B [1] as a state-based modelling language, mainly because of the *effective refinement* of models: an abstract model expressing the requirements of a given system can be verified and validated easily; a concrete model corresponding to the actual system is *constructed* incrementally and progressively by *refining* the abstraction. Event-B is also supported by a complete toolset RODIN [24] providing features like refinement, proof obligations generation, proof assistants and model-checking facilities.

The EVENT B modelling language can express *safety properties*, which are either *invariants* or *theorems* in a model corresponding to the system. Two main structures are available in EVENT B : **(1)** Contexts express static informations about the model; **(2)** Machines express dynamic informations about the model, safety properties, and events. An EVENT B model is defined *either* as a context or as a machine. A machine organises events (or actions) modifying state variables and uses static informations defined in a context. These basic structures are extended by the *refinement of models* which *relates* an *abstract* model and a *concrete* model.

Modelling Actions Over States. An EVENT B model is characterised by a (finite) list x of *state variables* possibly modified by a (finite) list of *events*. An invariant $I(x)$ states properties that must always be satisfied by the variables x and *maintained* by the activation of the events. The general form of an event e is as follows: ANY t WHERE $G(t,x)$ THEN $x :| P(t,x,x')$ END and corresponds to the transformation of the state of the variable x, which is described by a *before-after* predicate $BA(e)(x,x')$: the predicate is semantically equivalent to $\exists t \cdot G(t,x) \wedge P(t,x,x')$ and expresses the relationship linking the values of the state variables before (x) and just after (x') the *execution* of the event e. Proof obligations are produced by RODIN, from events: INV1 and INV2 state that an invariant condition $I(x)$ is preserved; their general form follows immediately from the definition of the before-after predicate $BA(e)(x,x')$ of each event e; FIS expresses the feasibility of an event e, with respect to the invariant I. By proving feasibility, we achieve that $BA(e)(x,z)$ provides a next state whenever the guard $grd(e)(x)$ holds: the guard is the enabling condition of the event.

INV1	INV2	FIS
$Init(x) \Rightarrow I(x)$	$I(x) \wedge BA(e)(x,x') \Rightarrow I(x')$	$I(x) \wedge grd(e)(x) \Rightarrow \exists z \cdot BA(e)(x,z)$

Model Refinement. The refinement of models extends the structures described previously, and relates an abstract model and a concrete model. This feature allows users to develop EVENT B models gradually and validate each decision step using the proof tool. The refinement relationship is expressed as follows: a model *AM* is refined by a model *CM*, when *CM simulates AM* (i.e. when a concrete event *ce* occurs in *CM*, there must be a corresponding enabling abstract event *ae* in *AM*). The final concrete model is closer to the behaviour of a real system that observes events using real source code. The relationships between contexts, machines and events are illustrated by the following diagrams (Fig.1) , which consider refinements of events and machines.

$$\begin{tikzcd}
I(x) & x \arrow[r, "\text{ae}"] \arrow[d, leftrightarrow] & x' \arrow[d, leftrightarrow] & I(x') \\
J(x,y) & y \arrow[r, "\text{ce}"] & y' & J(x',y')
\end{tikzcd}
\qquad
\begin{tikzcd}
I(x) & AM \arrow[r, "\text{SEES}"] & AC & \mathcal{T}h_i \\
J(x,y) & CM \arrow[r, "\text{SEES}"] \arrow[u, "REFINES"] & CC \arrow[u, "EXTENDS"'] & \mathcal{T}h_{i+1}
\end{tikzcd}$$

Fig. 1. Machines and Contexts relationships

The refinement of a formal model allows us to enrich the model via a *step-by-step* approach and is the foundation of our *correct-by-construction* approach [15]. Refinement provides a way to strengthen invariants and to add details to a model. It is also used to transform an abstract model to a more concrete version by modifying the state description. This is done by extending the list of state variables (possibly suppressing some of them), by refining each abstract event to a set of possible concrete versions, and by adding new events.

We suppose (see Fig.1) that an abstract model AM with variables x and an invariant $I(x)$ is refined by a concrete model CM with variables y. The abstract state variables, x, and the concrete ones, y, are linked together by means of a, so-called, gluing invariant $J(x,y)$. Event ae is in abstract model AM and event ce is in concrete model CM. Event ce refines event ae. $BA(ae)(x,x')$ and $BA(ce)(y,y')$ are predicates of events ae and ce respectively; we have to discharge the following proof obligation:

$$I(x) \land J(x,y) \land BA(ce)(y,y') \Rightarrow \exists x' \cdot (BA(ae)(x,x') \land J(x',y'))$$

We have briefly introduced the EVENT B modelling language and the structures proposed for organising the development of state-based models. In fact, the refinement-based development of EVENT B requires a very careful derivation process, integrating possible *tough* interactive proofs for discharging generated proof obligations, at each step of development.

3 State Properties and Refinement Diagrams

This section extends semantically EVENT B and introduces a way to deal with liveness properties using especially the refinement diagrams and the *leads to* ($\leadsto$) operator. Refinement diagrams have been introduced in a previous work [19], in order to help to develop sequential programs using refinement. The notation using the *leads to* operator $A \leadsto B$ is defined by the temporal assertion "$\Box(A \Rightarrow \Diamond B)$". This formula means that every A will *eventually* be followed by B.

Extending the Scope of EVENT B Properties. EVENT B allows users to express safety properties on models considered as reactive systems. An EVENT B model is valid with respect to a set of discharged proof obligations. However, since we have a list of events for each model, we can simulate reactions to events by extending the semantical scope of EVENT B properties. This extension of the properties taken into account by EVENT B to liveness ones, requires the definition of traces for an EVENT B model in an operational style. Therefore, we propose the use of the TLA [14] framework to support our proofs, as the framework provides simple temporal modalities, such as liveness and fairness.

We first define the temporal framework of an EVENT B machine M, using the following TLA notations: $Init$ is the predicate specifying initial states; $\Box[Next]_y$ means that each pair of consecutive states either satisfies $Next$ or leaves the values of y unchanged; $\mathrm{WF}_y(Next)$ expresses a *weak fairness* condition over $Next$.

Definition 1. *Let M be an* EVENT B *machine and C a context seen by M. Let y be the list of variables of M, let E be the set of events of M, and let $Init(y)$ be the predicate defining the initial values of y in M. The temporal framework of M is defined by the TLA specification $\mathcal{S}pec(M)$: $Init(y) \wedge \Box[Next]_y \wedge \mathrm{WF}_y(Next)$, where $Next \equiv \exists e \in E.BA(e)(y,y')$.*

Following Lamport [14], $\mathcal{S}pec(M)$ is valid for the set of infinite traces simulating M, with respect to the events of M and to fairness constraints. The set of traces for M is a subset of $Values^{\omega}$, which is the set of infinite words over the set of possible values of y in M, namely $Values$.

Liveness properties for M are, de facto, defined in TLA as follows. M satisfies $P \leadsto Q$ when $\Gamma(M) \vdash \mathcal{S}pec(M) \Rightarrow (P \leadsto Q)$. $\Gamma(M)$ is the proof context of M. Obviously, safety properties can be reformulated in the same framework. As for liveness properties, we can also use the *wp*-based approach for defining these properties under weak fairness. We can apply as well the works of Abrial et al [9, 18] on mathematical semantics in a *wp* framework, and on specific constructs [4] to state liveness properties as events.

Refinement Diagrams and *Leads To* ($\leadsto$) Operator. *Refinement diagrams* are used to develop the machine M and to add control in the EVENT B models. These diagrams are close to predicate diagrams [8] and to proof lattices introduced by Owicki and Lamport in [22] for representing (proofs of) liveness properties under fairness assumptions. We do not use these diagrams for proving but for supporting refinement. We construct the refinement lattices by applying the inference rules for the temporal operator *leads to* ($\leadsto$).

Definition 2. *Let M be an* EVENT B *machine and C a context seen by M. A is a set of assertions; $I(M)$ is the invariant of M; c are (control) variables of M, with values identifying the control points of M (e.g. start, end, etc.); G is a finite set of assertions for M called conditions of the form $g(x)$, where x are variables of M. Let E be the set of events for M.*

A refinement diagram for M, over A, is a labeled directed graph over A, with labels from G or E, satisfying the following rules:

- *If R is related to S by a unique arrow labeled $e \in E$, then*
  - *It satisfies the property $R \leadsto S$*
  - $\forall c,x,c',x'.R(c,x) \wedge I(M)(c,x) \wedge BA(e)(c,x,c',x') \Rightarrow S(c',x')$
  - $\forall c,x.R(c,x) \wedge I(M)(c,x) \Rightarrow \exists c',x'.BA(e)(c,x,c',x')$
- *If R is related to $S_1, \ldots, S_p$, then*
  - *Each arrow R to S_i is labeled by a guard $g_i \in G$.*
  - *For any i in $1..p$ the following conditions hold.*
    $\begin{pmatrix} R \wedge I(M) \wedge g_i(x) \Rightarrow S_i \\ \forall j.j \in 1..p \wedge j \neq i \wedge R \wedge I(M) \wedge g_i(x) \Rightarrow \neg g_j(x) \end{pmatrix}$
  - $R \wedge I(M) \Rightarrow \exists i \in 1..p.g_i$.

– *For each $e \in E$, there is only one instance of e in the diagram.*

A refinement diagram D for M, over A, is denoted by $PD(M) = (A, M, G, E)$.

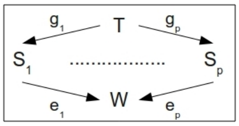


Fig. 2. A *refinement diagram*

A refinement diagram, as illustrated by the figure 2, relates a pair of assertions (T,W). We assume that T is a precondition, that can be decomposed into p assertions $S_1, \cdots, S_p$, and W is a postcondition.

Refinement diagrams can be used to infer the total correctness of an algorithm *constructed* step-by-step using refinement. The operator *leads to* ($\leadsto$) is transitive and confluent. Therefore, if a *refinement diagram* is built for a given problem, it is sound with respect to the requirements of the problem. *Refinement diagrams* possess proved properties [19], that we enumerate here.

Property 1. Let M be a machine and $D = (A, M, G, E)$ be a refinement diagram for M.

1. If M satisfies $P \leadsto Q$ and $Q \leadsto R$, it satisfies $P \leadsto R$.
2. If M satisfies $P \leadsto Q$ and $R \leadsto Q$, it satisfies $(P \vee R) \leadsto Q$.
3. If I is invariant for M and if M satisfies $P \wedge I \leadsto Q$, then M satisfies $P \leadsto Q$.
4. If I is invariant for M and if M satisfies $P \wedge I \Rightarrow Q$, then M satisfies $P \leadsto Q$.
5. If $P \xrightarrow{e} Q$ is a link of D for the machine M, then M satisfies $P \leadsto Q$.
6. If P and Q are two nodes of D such that there is a path in D from P to Q and any path from P can be extended in a path containing Q, then M satisfies $P \leadsto Q$.
7. If I, U, V, P, and Q are assertions such that I is the invariant of M; $P \wedge I \Rightarrow U$; $V \Rightarrow Q$; and there is a path from U to V and each path from U leads to V; then M satisfies $P \leadsto Q$.

These properties are derived from TLA definitions [14]. *Refinement diagrams* are a generalised version of diagrams proposed for developing sequential algorithms [19] and these are based on the *call-as-event* paradigm. Moreover, *refinement diagrams* are attached to EVENT B models and can be used for deriving liveness properties. The justification of such diagrams is based on the analysis of *leads to* properties and on liveness properties. The proof system of TLA contains proof rules for deriving the correctness of those properties. In the next section, we detail a paradigm for aiding the proof-based development of distributed algorithms.

4 *Service-As-Event* Paradigm

The EVENT B methodology requires skills in understanding the notion of *refinement*. Expertise is also required in the use of proof assistants and management of the modelling process, in order to ensure the discharging of proofs. In the EVENT B modelling method, the most important step is the expression of a very abstract definition of the problem to solve. The first abstract model usually gives a list of events corresponding to the *pre/post* specification with respect to the different cases. Each refinement step details progressively the abstract specifications (e.g. by decomposing them into *phases*, in the case of algorithmic systems, etc.). Each new step is checked by discharging proof obligations. Hence, the objective is clearly to simplify the effort of proof and explore simple ways to express a problem as a combination of (possibly reusable) *components*. We are interested in distributed algorithms; therefore, a *component* is equivalent to a *phase/step* of an algorithm. *Components* can be viewed as *"sub"*-distributed algorithms composing the actual algorithm.

In this paper, we present the *service-as-event* paradigm, inspired by the *call-as-event* [19] paradigm and based on *refinement diagrams*. The *service-as-event* paradigm helps us to state problems, using liveness properties, as for instance $P \leadsto Q$. An event e models the *effective* service leading from P to Q.

Primary Usage: Service Description. The *service-as-event* paradigm can help to state a problem in an abstract manner. The abstraction of a problem in Event B is as follows: An abstract event e expresses a *pre/post* specification. The *pre-condition* P is stated by the guard of the event e, whereas the *post-condition* Q is defined by the action of e. Using the properties of *refinement diagrams*, we can depict this statement with the property: $(P \xrightarrow{e} Q) \Rightarrow (P \leadsto Q)$. The event e expresses, in an abstract way, the service linking P and Q: every time P holds, e will be triggered and consequently, P will (*eventually*) be followed by Q.

Example 1. For instance, the leader election problem [2] is expressed using the following property: $acyclic(gr) \leadsto \exists rt,ts.\mathsf{spanning}(rt,ts,gr)$, where $acyclic(gr)$ states that gr is an acyclic connected graph and $spanning(rt,tr,gr)$ states that tr is a directed spanning tree of gr and its root rt is the leader. The property is illustrated by the *refinement diagram* 3 and simply stated in Event B , as follows:

$$\begin{tikzcd} acyclic(gr) \arrow[d, "\text{election}"] \\ \exists rt,ts.\mathsf{spanning}(rt,ts,gr) \end{tikzcd}$$

Fig. 3. A *refinement diagram* for leader election

```
EVENT election ≙
  WHEN
    $acyclic(gr)$
  THEN
    $rt,ts : |\mathsf{spanning}(rt',ts',gr)$
  END
```

The refinement diagram Fig.3 expresses that a process *election* is characterized by an abstract event election stating *what* is computed, but *not how* it is computed. The computation process is depicted in the refinement model, which will be defined later.

Extended Usage: Phase Identification from Service Decomposition. Another way to use the *service-as-event* paradigm is to decompose liveness properties, using the inference rules of the *leads to* ($\leadsto$) operator, such as the transitivity rule. In fact, we use the rules related to $\leadsto$ to break up a global service into multiple and simpler *"sub"*-services, analogous to steps or phases. As an illustration, one can decompose an Event B specification of a problem, represented by the property $P \leadsto Q$, as follows:

$$\dfrac{\dfrac{\dfrac{\cdots}{P \leadsto S}\,(\text{trans}) \quad \dfrac{\cdots}{S \leadsto R}\,(\text{trans})}{P \leadsto R}\,(\text{trans}) \qquad \dfrac{\dfrac{\cdots}{R \leadsto X}\,(\text{trans}) \quad \dfrac{\cdots}{X \leadsto Q}\,(\text{trans})}{R \leadsto Q}\,(\text{trans})}{P \leadsto Q}\,(\text{trans})$$

The initial property $P \leadsto Q$ is separated into several simpler properties (representing phases of the algorithm), until a satisfactory decomposition into independent phases, linked by *services* is obtained. Therefore, allowing the phases to be developed separately.

This process is similar to refinement, as shown by the following figure (see Fig.4): The first property $P \leadsto Q$ is associated with an abstract model, describing the service offered by the algorithm with an abstract event; the use of the transitivity rule to simplify $P \leadsto Q$ is interpreted as identifying the various steps from P to Q: it corresponds to the fact that new events, modelling intermediate transitions between P and Q, are added to the model, using refinement. A level in the proof tree is associated with a level of refinement. One may continue to decompose services, until each phase of the concrete algorithm is matched with a property.

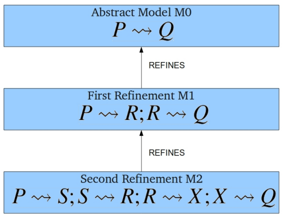

Abstract Model M0
$P \leadsto Q$
REFINES
First Refinement M1
$P \leadsto R; R \leadsto Q$
REFINES
Second Refinement M2
$P \leadsto S; S \leadsto R; R \leadsto X; X \leadsto Q$

Fig. 4. Refinement and Decomposition

Example 2. For instance, a routing algorithm can be decomposed into two phases: **(1)** a route discovery step, **(2)** a route maintenance/reconstruction, if the route is broken. Another point is to decompose the routing process into steps which are simpler, safer and more stable. In the next section, we give an example in which three phases for a routing algorithm are identified, to ensure that the routing service is satisfied.

First, the development methodology consists in decomposing a complex algorithm into simple fragments (services) using the *service-as-event* paradigm and *refinement diagrams*. The following step is to detail the developed services/phases and coordinate them by adding control. Hence, we can guide our refinement-base process by using refinement diagrams related to the EVENT B models.

5 Case Study: ANYCAST RP

We present in this section an example illustrating our modelling methodology (*refinement diagrams* and *service-as-event* paradigm), with the ANYCAST RP routing protocol [10, 11]. However, due to space requirements, we do not provide the whole development[1], we only give relevant details allowing us to explain clearly the methodology and the integration of models.

Introduction. ANYCAST RP is a protocol for multicast (one-to-many) communications (see Fig.5). In this protocol, a set of routers, called Designated Routers (*DR*), are used by directly connected sources to transmit data (*msg*) to another set of distant routers, the

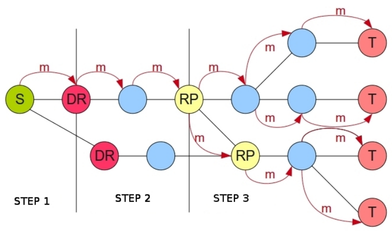


Fig. 5. ANYCAST RP

[1] Available at: `http://www.loria.fr/~andriami/ifm/index.html`

Rendezvous Points (*RP*). These Rendezvous Points (*RP*) are in charge of load sharing, redundancy and message delivery to connected destinations. ANYCAST RP is a non-toy protocol recommended by Cisco Systems, Inc as a reliable solution for multicasting [10]. Moreover, the protocol is cited as robust, scalable, having satisfactory bandwidth efficiency and good QoS [10, 13]. We use this protocol as an illustration, because it can be divided easily into *independent and sequential phases/steps*: the routing of messages (*msg*) **(1)** from sources (*s*) to Designated Routers (*DR*), **(2)** from Designated Routers (*DR*) to Rendezvous Points (*RP*) and **(3)** from Rendezvous Points (*RP*) to connected destinations. The following sections demonstrate the formal modelling of the ANYCAST RP protocol.

Abstract Model. We start with an abstract model ANYCAST_M0, describing the service offered by the protocol: that is, the routing and delivery of a message (*msg*), from a source (*s*) to a set of destinations (*g_t*). Sets of messages (*MESSAGES*), nodes (*NODES*), sources and destinations of each message (*m*) (indicated by functions *source* and *group_target*) are defined. Variables are also defined: *sent* contains messages sent by sources, *got* depicts messages received by destinations, *lost* contains lost messages. These variables are initialised with an empty set ($\varnothing$). Simple invariants constrain these variables: $got \cup lost \subseteq sent$; $got \cap lost = \varnothing$. Events define the behaviour of the system:

```
EVENT SENDING0
ANY
  $s, msg$
WHERE
  $s \in NODES$
  $msg \in MESSAGES$
  $msg \notin sent$
  $s = source(msg)$
THEN
  $sent := sent \cup \{msg\}$
```

```
EVENT RECEIVING0
ANY
  $g\_t, msg$
WHERE
  $g\_t \subseteq NODES$
  $msg \in MESSAGES$
  $msg \in sent$
  $msg \notin (got \cup lost)$
  $g\_t = group\_target(msg)$
THEN
  $got := got \cup \{msg\}$
```

```
EVENT RESENDING0
ANY
  $msg$
WHERE
  $msg \in MESSAGES$
  $msg \in sent$
THEN
  $lost := lost \setminus \{msg\}$
```

```
EVENT LOSING0
ANY
  $msg$
WHERE
  $msg \in MESSAGES$
  $msg \in sent$
  $msg \notin (got \cup lost)$
THEN
  $lost := lost \cup \{msg\}$
```

- We have events related to the protocol:
  - SENDING0 models the sending of a message (*msg*) by a source (*s*).
  - RECEIVING0 demonstrates the receiving of a message (*msg*) by a group of destinations (*g_t*): the message (*msg*) has been sent by a source, has not yet been lost nor received, therefore all the destinations (*g_t*) can receive the message (*msg*).
  - RESENDING0 depicts the re-sending of a message (*msg*): if the message (*msg*) has been lost, it is recovered.
- And events related to environment: LOSING0 presents the loss of a sent but not yet received message (*msg*).

This model is associated to the following *refinement diagram*:

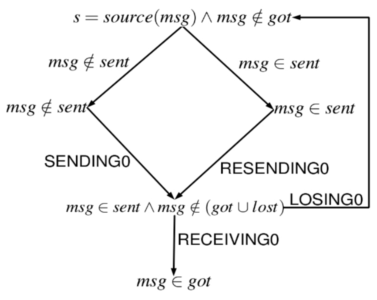


Fig. 6. Diagram $D0$ for ANYCAST_M0

The diagram $D0$ (see Fig.6) gives us the possibility to express the goal of ANYCAST RP as follows: $(s = source(msg) \wedge msg \notin got) \leadsto (msg \in got)$. The routing *service* allows a non-received message (msg), whose source is (s), to be eventually received by all of its destinations. This routing *service* can be decomposed into *sub-services*: a *sending* one (SENDING0), a *re-sending* one (RESENDING0) and a *receiving* one (RECEIVING0).

The service RECEIVING0 can be considered as the *main* service that allows users to verify the property $P \leadsto Q$ (with $P \ \widehat{=}\ (s = source(msg) \wedge msg \notin got)$ and $Q \ \widehat{=}\ (msg \in got)$), because it actually models the receiving of a message (msg) by the destinations ($got := got \cup \{msg\}$). The diagram $D0$ and the property $P \leadsto Q$ describe the normal behaviour of the algorithm, without errors. However, we also consider message losing in $P \leadsto F$, with $F \ \widehat{=}\ (msg \in lost)$, because messages can be lost (event LOSING0). But, since lost messages are sent again to the destinations (RESENDING0), and since we assume that the messages are not stuck in the *lost* state forever (to ensure progress of the algorithm), we have $F \leadsto Q$. Therefore, $P \leadsto Q$ is verified.

First Refinement. This refinement[2] (ANYCAST_M1) adds the Designated Routers (DR) between the sources (s) and the destinations (g_t). New variables are defined: dr_rcvd contains the messages received by some selected (not yet identified at this level of *abstraction*) Designated Routers (DR) from sources and dr_sent depicts the messages sent by selected Designated Routers (DR) to destinations; simple invariants are given: **(1)** $dr\_rcvd \subseteq sent$, **(2)** $dr\_sent \subseteq dr\_rcvd$, **(3)** $got \cup lost \subseteq dr\_sent$. Previous events are refined and new ones are added: DR_RECEIVING1 models the receiving of a message (msg) by a selected Designated Router (dr), from a source (s); DR_SENDING1 demonstrates the transmission of a message (msg) by a selected Designated Router (dr), to the destinations; DR_RESENDING1 is a refinement of RESENDING0. In fact, the sources are not in charge of the re-sending procedure, but the Designated Routers; RECEIVING1 presents the receiving of a message (msg) by destinations, from a Designated Router (dr); LOSING1 models losses of message (msg) between only Designated Routers and destinations, since losses between sources and Designated Routers are highly improbable. Let us denote by X the events DR_RESENDING1, RECEIVING1, LOSING1; and by Y their corresponding abstract versions: RESENDING0, RECEIVING0, LOSING0.

```
EVENT DR_RECEIVING1
WHEN
  $msg \in MESSAGES$
  $msg \in sent$
  $msg \notin dr\_rcvd$
THEN
  $dr\_rcvd := dr\_rcvd \cup \{msg\}$
```

```
EVENT DR_SENDING1
WHEN
  $msg \in MESSAGES$
  $msg \in dr\_rcvd$
  $msg \notin dr\_sent$
THEN
  $dr\_sent := dr\_sent \cup \{msg\}$
```

```
EVENT X REFINES Y
  ...
WHERE
  ...
  $\ominus$ $msg \in sent$
  $\oplus$ $msg \in dr\_sent$
  ...
```

[2] $\oplus$: to add an element to a model, $\ominus$: to remove an element from a model, ...: unchanged parts.

This model expresses an abstraction of ANYCAST RP, as follows: $P' \leadsto Q$, with $P' \mathrel{\widehat{=}} (s = source(msg) \wedge msg \notin sent)$ and $Q \mathrel{\widehat{=}} (msg \in got)$. We can see here that the initial predicate $P \mathrel{\widehat{=}} (s = source(msg) \wedge msg \notin got)$ is transformed into P', which is more detailed and more precise, saying that the message *msg* is not received because it has not yet been sent. An additional step related to the Designated Routers is added: $P' \leadsto R \wedge R \leadsto Q$, with an intermediate step R being $msg \in dr\_rcvd$.

Decomposing ANYCAST RP into Phases. The model ANYCAST_M2 introduces another intermediate routing: messages must be redirected to their destinations by routers called Rendezvous Points (*RP*). New variables are added in this refinement: *rp_rcvd* represents the messages received by some selected (not yet identified at this level of *abstraction*) Rendezvous Points (*RP*) from Designated Routers (*DR*) and *rp_sent* depicts the messages sent by selected Rendezvous Points (*RP*) to destinations; simple invariants on these variables are defined: **(1)** $rp\_rcvd \subseteq dr\_sent$, **(2)** $rp\_sent \subseteq rp\_rcvd$, **(3)** $got \subseteq rp\_sent$, **(4)** $got \subseteq rp\_rcvd$, **(5)** $rp\_rcvd \cap lost = \varnothing$. The last invariant describes an assumption on the system: messages can only be lost between selected Designated Routers (*DR*) and Rendezvous Points (*RP*). Events are refined or added: RP_RECEIVING2 models the receiving of a message (*msg*) by a selected Rendezvous Point, from a Designated Router; RP_SENDING2 presents the sending of a message (*msg*) by selected Rendezvous Point to destinations; RECEIVING2 depicts the receiving of a message (*msg*) by all the destinations of the message; LOSING2 models the losses of messages between Designated Routers (*DR*) and Rendezvous Points (*RP*).

```
EVENT RP_RECEIVING2
ANY
  $msg$
WHERE
  $msg \in MESSAGES$
  $msg \in dr\_sent$
  $msg \notin lost$
THEN
  $rp\_rcvd := rp\_rcvd \cup \{msg\}$
```

```
EVENT RP_SENDING2
ANY
  $msg$
WHERE
  $msg \in MESSAGES$
  $msg \in rp\_rcvd$
THEN
  $rp\_sent :=$
     $rp\_sent \cup \{msg\}$
```

```
EVENT RECEIVING2
  ...
WHERE
$\ominus$ $msg \in dr\_sent$
$\ominus$ $msg \notin (got \cup lost)$
$\oplus$ $msg \in rp\_sent$
$\oplus$ $msg \notin got$
  ...
```

```
EVENT LOSING2
  ...
WHERE
  ...
$\ominus$ $msg \notin (got \cup lost)$
$\oplus$ $msg \notin rp\_rcvd$
$\oplus$ $msg \notin lost$
  ...
```

This model defines the entire dataflow that occurs during ANYCAST RP: **(1)** from sources to Designated Routers (*DR*), **(2)** from Designated Routers (*DR*) to Rendezvous Points (*RP*) and **(3)** from Rendezvous Points (*RP*) to destinations. This description of the complete dataflow is emphasized by the *refinement diagram* of the model:

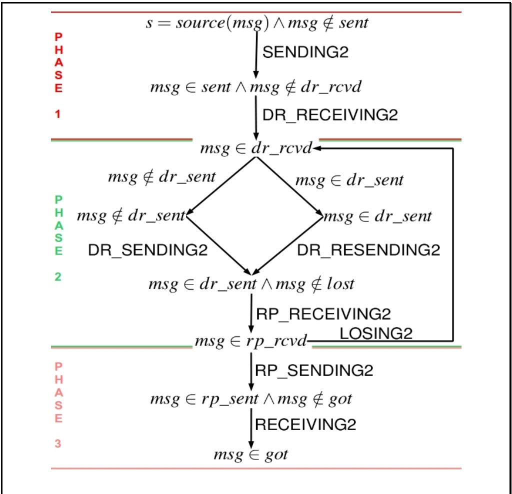


Fig. 7. Diagram $D2$ for ANYCAST_M2

The diagram $D2$ (see Fig.7) allows us to express ANYCAST RP using the following property: $P' \leadsto R \land R \leadsto S \land S \leadsto Q$, with $P' \mathrel{\widehat{=}} (s = source(msg) \land msg \notin sent)$, $R \mathrel{\widehat{=}} (msg \in dr\_rcvd)$, $S \mathrel{\widehat{=}} (msg \in rp\_rcvd)$, and $Q \mathrel{\widehat{=}} (msg \in got)$. We have decomposed $R \leadsto Q$, by transitivity, to add a new step related to the additional routing (Rendezvous Points). Moreover, the diagram allows us to identify phases of the protocol: **(Phase 1)** Routing from sources to Designated Routers, **(Phase 2)** Routing from Designated Routers to Rendezvous Points, **(Phase 3)** Routing from Rendezvous Points to Destinations.

The three identified phases are independent and can be developed separately: we have **(1)** $P' \leadsto R$, **(2)** $R \leadsto S$ and **(3)** $S \leadsto Q$. The development in phases is driven by the location of a message/packet in the network and by the type of nodes.

Combining and Coordinating Phases. We have divided ANYCAST RP into three *components*, described by the diagram $D2$ (see Fig.7), in the previous section. Since we develop the components independently, we relax the conditions $msg \in dr\_rcvd$, $msg \in rp\_rcvd$. We replace them by $msg \in MESSAGES$. The goal is to reintroduce these conditions (or their refined forms) in the events to add control and coordination during the combination of phases. Abstract models of each phase are composed of the events of ANYCAST_M2 related to **(1)** $P' \leadsto R$, **(2)** $R \leadsto S$ and **(3)** $S \leadsto Q$ and modified as said previously (relaxing some conditions). The next paragraphs give the development of each phase.

Phase 1: From sources to Designated Routers. We introduce the identity of a selected Designated Router (dr), which receives a message (msg) sent by a source (s). We notice that the variables $sent$, dr_rcvd have been replaced by $msg\_sent\_by\_src$ and $msg\_rcvd\_by\_dr$. These variables associate sent/received messages with the identities of senders (sources) and receivers (Designated Routers).

```
EVENT SENDING
ANY
  $s, msg, dr, dest\_grp$
WHERE
  $s \in SOURCES \land msg \in MESSAGES$
  $dest\_grp \subseteq DR \land dr \in DR \land s = source(msg)$
  $dest\_grp = \{dr\} \land s \mapsto msg \notin msg\_sent\_by\_src$
THEN
  $msg\_sent\_by\_src := msg\_sent\_by\_src \cup \{s \mapsto msg\}$
  $dr\_dest := dr\_dest \cup (dest\_grp \times \{msg\})$
```

```
EVENT DR_RECEIVING
ANY
  $dr, msg$
WHERE
  $dr \in DR \land msg \in MESSAGES$
  $dr \mapsto msg \in dr\_dest$
  $dr \mapsto msg \notin msg\_rcvd\_by\_dr$
THEN
  $msg\_rcvd\_by\_dr := msg\_rcvd\_by\_dr \cup \{dr \mapsto msg\}$
```

SENDING models the sending of a message (*msg*) by a source (*s*) to a Designated Router (*dr*). A variable *dest_grp* indicates the Designated Router (*dr*) target of a message (*msg*). RECEIVING presents the receiving of a message (*msg*) by a Designated Router (*dr*).

Phase 2: From Designated Routers to Rendezvous Points. This model identifies the selected Designated Router (*dr*), sender of a message (*msg*) and the chosen Rendezvous Point (*rp*), target of the message.

```
EVENT DR_SENDING
ANY
  dr, rp, msg, dest_grp
WHERE
  dr ∈ DR ∧ rp ∈ RP ∧ dest_grp ⊆ RP
  dest_grp = {rp} ∧ msg ∈ MESSAGES
  msg ∉ ran(msg_sent_by_dr)
THEN
  msg_sent_by_dr := msg_sent_by_dr ∪ {dr ↦ msg}
  rp_dest := rp_dest ∪ (dest_grp × {msg})
```

```
EVENT LOSING
ANY
  msg, dr, rp
WHERE
  dr ∈ DR ∧ rp ∈ RP ∧ msg ∈ MESSAGES
  dr ↦ msg ∈ msg_sent_by_dr
  rp ↦ msg ∈ rp_dest ∧ msg ∉ lost
  rp ↦ msg ∉ msg_rcvd_by_rp
THEN
  lost := lost ∪ {msg}
```

```
EVENT DR_RESENDING
ANY
  msg, dr, rp
WHERE
  dr ∈ DR ∧ rp ∈ RP ∧ msg ∈ MESSAGES
  dr ↦ msg ∈ msg_sent_by_dr ∧ rp ↦ msg ∈ rp_dest
THEN
  lost := lost \ {msg}
```

```
EVENT RP_RECEIVING
ANY
  rp, msg
WHERE
  rp ∈ RP ∧ msg ∈ MESSAGES
  rp ↦ msg ∈ rp_dest ∧ msg ∉ lost
THEN
  msg_rcvd_by_rp := msg_rcvd_by_rp ∪ {rp ↦ msg}
```

We use the same techniques as in phase 1 to identify the senders and receivers of a message, namely Designated Routers and Rendezvous Points: the variables *dr_sent*, *rp_rcvd* are replaced with *msg_sent_by_dr* and *msg_rcvd_by_rp*, which associate sent/received messages with the identities of senders and receivers; *rp_dest* indicates the selected Rendezvous Points destinations of sent messages.

Phase 3: From Rendezvous Points to Destinations. The identities of selected Rendezvous Points (*rp*) sending messages (*msg*) to destinations are introduced by this model.

```
EVENT RP_SENDING
ANY
  rp, msg
WHERE
  rp ∈ RP ∧ msg ∈ MESSAGES
THEN
  msg_sent_by_rp := msg_sent_by_rp ∪ {rp ↦ msg}
```

```
EVENT RECEIVING
ANY
  gt, msg
WHERE
  gt ⊆ TARGETS ∧ msg ∈ MESSAGES ∧ msg ∉ got
  gt = group_target(msg) ∧ msg ∈ ran(msg_sent_by_rp)
THEN
  got := got ∪ {msg}
```

This model is simple to understand: a Rendezvous Points (*rp*) sends (RP_SENDING) a message (*msg*) to a group of destinations (*g_t*), which receive the message (RECEIVING).

We show how the uses of *refinement diagrams* and the *service-as-event* paradigm help in the models (*components*) combination and coordination. First, we draw the *refinement diagrams* for each phase, and then, we add/combine predicates to link the diagrams and models. The *diagram D* (see Fig.8) shows three *sub-diagrams* for each phase, and demonstrates how the phases can be coordinated to obtain a formal model of ANYCAST RP: to link two consecutive phases, we form a conjunction with the *post-condition* of the first phase and the *pre-condition* of the other one; for example, to combine phases 1 and 2, we use a property resulting of the conjunction of the *pre/post*:

$dr \in DR \wedge dest\_grp = \{rp\} \wedge dr \mapsto msg \in msg\_rcvd\_by\_dr$, meaning that only a message (msg), received by a Designated Router (dr), can be sent to a Rendezvous Point (rp).

The same applies for the combination of phases 2 and 3, as we also use the result of the conjunction of the *pre/post*: $rp \mapsto msg \in msg\_rcvd\_by\_rp$. We notice that in fact, the combination is equivalent here to the fact of linking and ordering *sending* and *receiving services*, e.g. $sending1 \leadsto receiving1 \wedge receiving1 \leadsto sending2 \wedge \ldots$ These operations on *refinement diagrams* are reflected in the resulting model after integrating phases.

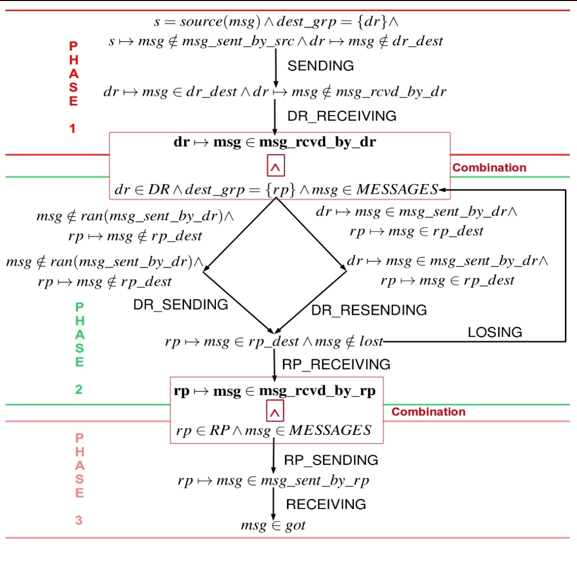

$s = source(msg) \wedge dest\_grp = \{dr\} \wedge$
$s \mapsto msg \notin msg\_sent\_by\_src \wedge dr \mapsto msg \notin dr\_dest$
PHASE 1
SENDING
$dr \mapsto msg \in dr\_dest \wedge dr \mapsto msg \notin msg\_rcvd\_by\_dr$
DR_RECEIVING
$dr \mapsto msg \in msg\_rcvd\_by\_dr$
$\wedge$
Combination
$dr \in DR \wedge dest\_grp = \{rp\} \wedge msg \in MESSAGES$
$msg \notin ran(msg\_sent\_by\_dr) \wedge$
$rp \mapsto msg \notin rp\_dest$
$dr \mapsto msg \in msg\_sent\_by\_dr \wedge$
$rp \mapsto msg \in rp\_dest$
$msg \notin ran(msg\_sent\_by\_dr) \wedge$
$rp \mapsto msg \notin rp\_dest$
$dr \mapsto msg \in msg\_sent\_by\_dr \wedge$
$rp \mapsto msg \in rp\_dest$
DR_SENDING
DR_RESENDING
PHASE 2
LOSING
$rp \mapsto msg \in rp\_dest \wedge msg \notin lost$
RP_RECEIVING
$rp \mapsto msg \in msg\_rcvd\_by\_rp$
$\wedge$
Combination
$rp \in RP \wedge msg \in MESSAGES$
PHASE 3
RP_SENDING
$rp \mapsto msg \in msg\_sent\_by\_rp$
RECEIVING
$msg \in got$

Fig. 8. Diagram *D* for Phases Combination

A set of invariants related to ordering and coordinating for *sending* and *receiving* services is added to the model: $\mathbf{msg\_sent\_by\_dr} \subseteq \mathbf{msg\_rcvd\_by\_dr}$ and $\mathbf{msg\_sent\_by\_rp} \subseteq \mathbf{msg\_rcvd\_by\_rp}$ express that only received messages are sent by respectively selected Designated Routers and Rendezvous Points. According to the diagram 8, the *pre-conditions* of the *sending* services DR_SENDING, RP_SENDING must also be modified. Therefore, we propose to modify these events as follows, by strengthening their guards: We add new guards that state that *receiving* services have to occur before following *sending* ones.

```
EVENT DR_SENDING
  ...
WHERE
  ...
  ⊕ dr ↦ msg ∈ msg_rcvd_by_dr
  ...
```

```
EVENT RP_SENDING
  ...
WHERE
  ...
  ⊕ rp ↦ msg ∈ msg_rcvd_by_rp
  ...
```

The model expresses the following property (see Fig.8), which describes ANYCAST RP: $P \leadsto R \quad \wedge \quad R \leadsto S \quad \wedge \quad S \leadsto Q$, with $P \;\widehat{=}\; (s = source(msg) \wedge dest\_grp = \{dr\} \wedge s \mapsto msg \notin msg\_sent\_by\_src \wedge dr \mapsto msg \notin dr\_dest)$, $R \;\widehat{=}\; (dr \in DR \wedge dest\_grp = \{rp\} \wedge dr \mapsto msg \in msg\_rcvd\_by\_dr)$, $S \;\widehat{=}\; (rp \mapsto msg \in msg\_rcvd\_by\_rp)$, $Q \;\widehat{=}\; (msg \in got)$. We refine this model, until we obtain a *local model* (where events are local to nodes of the network), from which an algorithmic form of the protocol can be derived. An interesting property of these kinds of combination is that one can develop the phases separately and choose at which level of refinement the combination will occur. Moreover, the splitting of ANYCAST RP into small pieces helped us to concentrate our main efforts on finding correct ways of composing and coordinating the models of the phases, understanding and discharging the proof obligations generated by the integration of models.

6 Discussion, Conclusion and Future Work

We have introduced the *service-as-event* paradigm, as an integration of the EVENT B language with temporal notations and diagrams to cope with liveness properties, system decomposition and *components* integration, and as an extension of the *call-as-event* paradigm [19]. The diagrammatic notation describing *services*, namely *the refinement diagrams*, provides a graphical mean to support the intuition. These diagrams are particularly suited for guiding refinement-based development, because their refinement is possible [8]. The underlying semantical framework behind them is based on trace semantics and temporal structures, derived from TLA. In the literature, Manna and Pnueli [16] developed a collection of verification techniques based on *verification diagrams*, which are related to proving various temporal properties (invariance, safety, fairness, liveness, etc.) of reactive systems. They introduced different diagrams (WAIT-FOR and INVARIANCE diagrams, CHAIN diagrams, etc.), which are related to proof rules for deriving these properties. Our refinement diagrams are similar but we use them for refinement and they are integrated to the EVENT B models; the objectives are clearly to help in the refinement of complex systems and to decompose systems into subsystems in a *correct-by-construction* process. UNITY [9, 18] proposes also a combination of temporal logic and actions systems using the superposition technique, and a modelling of distributed and parallel programs under weak fairness, which is a limitation for expressing general fairness assumptions.

Our case study (ANYCAST RP) is simple to understand, because the protocol contains three identifiable, consecutive and independent routing phases, expressed as follows: $msg \notin sent \leadsto msg \in sent \land msg \in sent \leadsto msg \in received$. This simplicity hides technical details of the Event B models of the phases. In fact, the decomposition of the (complex) problem into smaller sub-problems allows us to discharge easy proof obligations related to parts of the algorithm and helps us to focus our efforts on the integration of models. However, decompositions of systems may present more difficulties and require a clever analysis. The application of the *service-as-event* paradigm and *refinement diagrams* is effective for modelling distributed algorithms with behaviours that can be decomposed into strict, sequential and/or non-interfering (or with little interferences) phases local to nodes: we have solved other case studies, related to Network-on-Chips [5], especially the *XY routing* and *network dynamic reconfiguration* services.

Our future works involve the connection of our approach to a platform integrating real concurrency concepts related to effective programming languages based on the *service-as-event* paradigm. Moreover, we plan to delve into the topic of feature interactions and how interferences can be taken into account. Another point is the relation between the complexity of proofs and models reuse for the description of the routing phases. The reuse and the adaptation of formal models are related to formal design patterns [3]. Finally, we intend to develop few more case studies related to distributed networks, and our goal is to develop a toolbox that can be used to implement distributed protocols using a programming language, where the toolbox will transform the verified formal specifications of EVENT B models [5, 6] into a given programming language.

References

1. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering (2010)
2. Abrial, J.-R., Cansell, D., Méry, D.: A mechanically proved and incremental development of ieee 1394 tree identify protocol. Formal Asp. Comput. 14(3), 215–227 (2003)
3. Abrial, J.-R., Hoang, T.S.: Using design patterns in formal methods: An event-B approach. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 1–2. Springer, Heidelberg (2008)
4. Abrial, J.-R., Mussat, L.: Introducing Dynamic Constraints in B. In: B98, pp. 83–128 (1998)
5. Andriamiarina, M.B., Daoud, H., Belarbi, M., Méry, D., Tanougast, C.: Formal Verification of Fault Tolerant NoC-based Architecture. In: First International Workshop on Mathematics and Computer Science (IWMCS 2012), Tiaret, Algérie (December 2012)
6. Andriamiarina, M.B., Méry, D., Singh, N.K.: Revisiting Snapshot Algorithms by Refinement-based Techniques. In: PDCAT, IEEE Computer Society (2012)
7. Back, R.-J., Sere, K.: Stepwise refinement of action systems. Structured Programming 12(1), 17–30 (1991)
8. Cansell, D., Méry, D., Merz, S.: Diagram refinements for the design of reactive systems. J. UCS 7(2), 159–174 (2001)
9. Chandy, K.M., Misra, J.: Parallel Program Design A Foundation. Addison-Wesley Publishing Company (1988) ISBN 0-201-05866-9
10. Cisco Systems. Anycast RP, `http://www.cisco.com/en/US/docs/ios/solutions_docs/ip_multicast/White_papers`
11. Cisco Systems. Anycast RP using PIM, `http://tools.ietf.org/html/draft-ietf-pim-anycast-rp-07`
12. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (2000)
13. Kang, J., Sucec, J., Kaul, V., Samtani, S., Fecko, M.A.: Robust pim-sm multicasting using anycast rp in wireless ad hoc networks. In: Proceedings of the 2009 IEEE International Conference on Communications, ICC 2009, pp. 5139–5144. IEEE Press, Piscataway (2009)
14. Lamport, L.: A temporal logic of actions. ACM Trans. Prog. Lang. Syst. 16(3), 872–923 (1994)
15. Leavens, G.T., Abrial, J.-R., Batory, D.S., Butler, M.J., Coglio, A., Fisler, K., Hehner, E.C.R., Jones, C.B., Miller, D., Jones, S.L.P., Sitaraman, M., Smith, D.R., Stump, A.: Roadmap for enhanced languages and methods to aid verification. In: Jarzabek, S., Schmidt, D.C., Veldhuizen, T.L. (eds.) GPCE, pp. 221–236. ACM (2006)
16. Manna, Z., Pnueli, A.: Temporal verification diagrams. In: Hagiya, M., Mitchell, J.C. (eds.) TACS 1994. LNCS, vol. 789, pp. 726–765. Springer, Heidelberg (1994)
17. McIver, A., Morgan, C.: Abstraction, Refinement And Proof For Probabilistic Systems (Monographs in Computer Science). Springer (2004)
18. Méry, D.: Requirements for a temporal B: Assigning Temporal Meaning to Abstract Machines. and to Abstract Systems. In: Galloway, A., Taguchi, K. (eds.) IFM 1999 Integrated Formal Methods 1999, YORK (June 1999)
19. Méry, D.: Refinement-based guidelines for algorithmic systems. Int. J. Software and Informatics 3(2-3), 197–239 (2009)
20. Méry, D., Singh, N.K.: Analysis of DSR protocol in event-B. In: Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems, SSS 2011, pp. 401–415. Springer-Verlag, Heidelberg (2011)

21. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Informatica 6, 319–340 (1976)
22. Owicki, S., Lamport, L.: Proving liveness properties of concurrent programs. ACM Trans. Program. Lang. Syst. 4(3), 455–495 (1982)
23. Owre, S., Shankar, N.: A brief overview of PVS. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 22–27. Springer, Heidelberg (2008)
24. Project RODIN. Rigorous open development environment for complex systems (2004-2010), `http://www.eventb.org/`
25. Rehm, J., Cansell, D.: Proved Development of the Real-Time Properties of the IEEE 1394 Root Contention Protocol with the Event B Method. In: ISoLA, pp. 179–190 (2007)
26. Tanenbaum, A.S.: Computer networks (4. ed.). Prentice-Hall (2002)
27. Wenzel, M., Paulson, L.C., Nipkow, T.: The Isabelle Framework. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 33–38. Springer, Heidelberg (2008)

Quantified Abstractions of Distributed Systems

Elvira Albert[1], Jesús Correas[1], Germán Puebla[2], and Guillermo Román-Díez[2]

[1] DSIC, Complutense University of Madrid (UCM), Spain
[2] DLSIIS, Technical University of Madrid (UPM), Spain

Abstract. When reasoning about distributed systems, it is essential to have information about the different kinds of nodes which compose the system, how many instances of each kind exist, and how nodes communicate with other nodes. In this paper we present a static-analysis-based approach which is able to provide information about the questions above. In order to cope with an unbounded number of nodes and an unbounded number of calls among them, the analysis performs an *abstraction* of the system producing a graph whose nodes may represent (infinitely) many concrete nodes and arcs represent any number of (infinitely) many calls among nodes. The crux of our approach is that the abstraction is enriched with upper bounds inferred by a *resource analysis* which limit the number of concrete instances which the nodes and arcs represent. The combined information provided by our approach has interesting applications such as debugging, optimizing and dimensioning distributed systems.

1 Introduction

When reasoning about distributed systems, it is essential to have information about their *configuration*, i.e., the sorts and quantities of nodes which compose the system, and their *communication*, i.e., with whom and how often the different nodes interact. Whereas configurations may be straightforward in simple applications, the tendency is to have rather complex and dynamically changing configurations. Cloud computing [5] is an example of this. In this paper, we introduce the notion of *Quantified Abstraction* (QA for short) of a distributed system which abstracts both its configuration and communication by means of static analysis. QAs are *abstract* in the sense that a single abstract node may represent (infinitely) many nodes and a single abstract interaction may represent (infinitely) many interactions. QAs are *quantified* in that we provide an upper bound on the (possibly infinite) number of actual nodes which each abstract node represents, and an upper bound on the (possibly infinite) number of actual interactions which each abstract interaction represents. Note that abstraction allows dealing with an unbounded number of elements in the system, whereas the upper bounds allow regaining accuracy by bounding the number of elements which each abstraction represents.

Actors form a well established model for distributed systems [14,4,6,12]. We apply our analysis to an Actor-like language [10] for distributed concurrent systems based on asynchronous communication. The distribution model is based on (possibly interacting) objects which are grouped into distributed *nodes*, called *coboxes*. Objects belong to their corresponding cobox for their entire lifetime. To realize concurrency, each cobox supports multiple, possibly interleaved, processes which we refer to as *tasks*. Tasks are created when methods are asynchronously called on objects, e.g., $o!m()$ starts a new task. The callee object o is responsible for executing the method call. The communication can be observed by tracking the calls between each pair of objects (e.g., we have a communication between the *this* object and o due to the invocation of m). Informally, given an execution, its *configuration* consists of the set of coboxes which have been created along such execution and which are the nodes of the distributed system, together with the set of objects created within each cobox. Similarly, the *communication* of an execution is defined as the set of calls between each pair of objects in the system; from which we can later obtain the communication for pairs of coboxes.

Statically inferring QAs is a challenging problem, since it requires (1) keeping track of the relations between the coboxes and the objects, (2) bounding the number of elements which are created, (3) bounding the number of interactions between objects, and (4) doing so in the context of distributed concurrent programming. The main contributions of this paper are:

1. *Abstract configurations*. The abstraction of objects and coboxes we rely on is based on *allocation sequences* [11] (i.e., the sequence of allocation sites where the objects that led to the creation of the current one were created). We use a points-to analysis to infer the allocation sequences which allow us to infer the ownership relations between the coboxes and the objects created.
2. *Quantified nodes*. We define a new cost model which can be plugged in the generic resource analyzer COSTABS [2] (without requiring any change to the analysis engine) in order to infer upper bounds on the number of coboxes and of objects that each element of an abstract configuration represents.
3. *Quantified edges*. We propose a cost model which can be also plugged in COSTABS to infer upper bounds on the number of calls among nodes.
4. *Implementation*. We have implemented our analysis in COSTABS and applied it on a case study developed by Fredhopper®. A notable result of our experiments is that COSTABS was able to spot an excessive number of connections between two distributed nodes that should be better allocated together.

QAs have many applications for optimizing, debugging and dimensioning distributed applications which include among others: (1) QAs provide a global view of the distributed application, which may help to detect errors related to the creation of the topology or task distribution. (2) They allow us to identify nodes that execute a too large number of processes while other siblings execute only a few of them. (3) They are required to perform meaningful resource analysis of distributed systems, since they allow determining to which node the computation of the different processes should be associated. (4) They allow us to detect components that have many interactions and that would benefit from being deployed in the same machine or at least have a very fast communication channel. (5) They provide a further step towards static bandwidth analysis.

2 The Language

We apply our analysis to the language ABS [10,12]. ABS extends the basic concurrent objects model [14,4,6,12] with the abstraction of object groups, named *coboxes*.

Each cobox conceptually has a dedicated processor and a number of objects can live inside the cobox and share its processor. Communication is based on asynchronous method calls with standard objects as targets. Consider an asynchronous method call m on object o, written as $f = o!m()$. The objects *this* and o communicate by means of the invocation m. Here, f is a future variable which allows synchronizing with the completion of task m by means of the await $f?$ instruction which behaves as follows. If m has finished, execution of the current task proceeds. Otherwise, the current task releases the processor to allow other available tasks (possibly a task of another object in the cobox) to take it. The language syntax is as follows. A *program* consists of a set of classes **class** C_1 $(t_1\ fn_1, ..., t_n\ fn_n)$ $\{M_1\ ...\ M_k\}$ where each $t_i\,fn_i$ declares a field fn_i of type t_i, and each M_i is a *method definition* $t\ m(t_1\ w_1, ..., t_n\ w_n)\ \{t_{n+1}\ w_{n+1}; \ldots; t_{n+p}\ w_{n+p};\ s\}$ where t is the type of the return value; $w_1, ..., w_n$ are the formal parameters with types $t_1, ..., t_n$; $w_{n+1}, ..., w_{n+p}$ are local variables with types $t_{n+1}, ..., t_{n+p}$; s is a sequence of instructions which adhere to the following grammar, where x and z denote standard variables, and y a future variable whose declaration includes the type of the returned value:

$s ::= in \mid in; s \quad b ::= e{>}e \mid e{==}e \mid b \wedge b \mid b \vee b \mid !b \quad e ::= \text{null} \mid \text{this}.f \mid x \mid e{+}e \mid e{*}e \mid e{-}e$
$in ::= x{=}\text{new}\ C(\bar{x}) \mid x{=}\text{newcog}\ C(\bar{x}) \mid x{=}e \mid \text{this}.f{=}e \mid y = x!m(\bar{z}) \mid \text{if}\ b\ \text{then}\ s_1\ \text{else}\ s_2 \mid$
$\quad \text{return}\ x \mid \text{while}\ b\ \text{do}\ s \mid \text{await}\ y?$

There is an implicit local variable called this that refers to the current object. Observe that the only fields which can be accessed are those of the current object, i.e., this. Thus, the language is data-race free [10], since no two tasks for the same object can be active at the same time. The instruction newcog (i.e., "new component object group") creates a new object, but instead of within the current cobox, the new object becomes the *root* of a brand new cobox. It is the root since all other objects which are transitively created using new belong to such new cobox, until other newcog instructions are executed, which introduce other coboxes with their respective roots. We assume all programs include a method called main, which does not belong to any class and has no fields, from which the execution starts in an implicitly created initial cobox, called ϵ.

Program execution is non-deterministic, i.e., given a state there may be different execution steps that can be taken, depending on the cobox selected and, when processors are released, it is also non-deterministic on the particular task within each cobox selected for further execution. We refer to [10] for a precise definition of the language semantics. For our purposes, we only need to know that a program state is formed by a set of coboxes, a set of objects and a set of futures. Each cobox simply contains a unique identifier and the identifier of the currently active object in the cobox (or $\emptyset$ if all objects are idle). Each object contains a unique identifier, the value of its fields, the method name of the active task, and a pool of suspended tasks. Each task in turn contains the values of the local variables and the list of instructions to execute. Execution steps are denoted $S \leadsto_l^b S'$, indicating that we move from state S to state S' by executing instruction b on the object identified by l. Traces take the form $t \equiv S_0 \leadsto_\epsilon^{b_0} \cdots \leadsto_{l_{n-1}}^{b_{n-1}} S_n$ where S_0 is an initial state in which only the main method is available.

Example 1. Our running example sketches an implementation of a distributed application to store and retrieve data from a database. The main method creates a new server and initializes it using two arguments, n, the number of *handlers* (i.e., objects that perform requests to the database), and m, the number of requests performed by each handler.

```
void main (Int n, Int m) {
① Server s = newcog Server(null);
   s!start (n,m);
}
class Server (DAO dao) {
  void start (Int n, Int m) {
    Fut f <void> = this!initDAO();
    await f?;
    while(n > 0) {
②     H h = new Handler(this.dao);
      h!run(m);
      n = n − 1;
    }
  }
  void initDAO () {
③   this.dao = new DAO(null);
    Fut f <void> = this.dao!initDB();
    await f?;
  }
}
```

```
class Handler (DAO dao) {
  void run (Int m) {
    while(m>0) {
      this.dao.query(m);
      m = m − 1;
    }
  }
}
class DAO (DB db) {
  void initDB () {
④   this.db = new DB();
  }
  boolean query(Int m) {
    String s = ... //query m
    this.db!exec(s);
  }
}
class DB () {
  boolean exec(String s) {...}
}
```

Method start initializes a data access object (DAO) that is used by Handler objects to request the database. Then, it creates n Handler objects at program point (p.p. for short) ② and starts their execution via the run method. The DAO object creates a fresh DB object at p.p. ④, that will actually execute queries from handlers. When executing run, each handler performs m requests to the DAO object by invoking method query. The use of Fut<void> variables and await instructions allow method synchronization. Regarding distribution, observe that the configuration contains a single distributed component (the Server cobox at ①), as all other objects are created using new. □

3 Background: Points-to and Resource Analysis

In this paper, we make use of the techniques of points-to analysis [11,13] and resource analysis [1,3] to infer quantified abstract configurations. We will try to use them as black boxes along the paper as much as possible. Still, we need to review the basic components that have to be used and/or adapted for our purposes.

3.1 Cost Centers and Points-to Analysis

An essential concept of the resource analysis framework for distributed systems in [1,3] is the notion of *cost center*. A cost center represents a distributed component (or node) of the system such that the cost performed on such component can be attributed to its cost center. Since in our language coboxes are the distributed components of the system, finding out the cost centers amounts to inferring the set of coboxes in the program. This can be done by means of points-to analysis [3]. The aim of points-to analysis is to approximate the set of objects (or coboxes) which each reference variable may point to during program execution. Following [11,13], the abstraction of each object created in the program is a syntactic construction of the form $o_{ij...pq}$, where all elements in $ij\ldots pq$ are allocation sites, which represents all run-time objects that were created at q when the enclosing instance method was invoked on an object represented by $o_{ij...p}$, which in turn was created at allocation site p. Let S be the set of all allocation sites in a program. Given a constant $k \geq 1$, the analysis considers a finite set of object names, denoted $\mathcal{N}$, which is defined as: $\mathcal{N} = \{\epsilon\} \cup S \cup S^2 \ldots S^k$. Note that k defines the maximum size of sequences of allocations, and it allows controlling the precision of the analysis. Allocation sequences have in principle unbounded length and thus it is sometimes necessary to lose precision during analysis. This is done by just keeping the k rightmost positions in sequences whose length is greater than k. We use $|s|$ to denote the length of a sequence s. We define the operation $\langle i, j, \ldots, p\rangle \oplus q$ for referring to the following object name: $o_{ij...pq}$ if $|\langle i, j, \ldots, p, q\rangle| \leq k$, or $o_{j...pq}$ otherwise. In addition, a variable can be assigned objects with different object names. In order to represent all possible objects pointed to by a variable, sets of object names are used. We will use the results of the points-to analysis by using $pt(q, x)$ which refers to the set of object names at p.p. q for a given reference variable x.

Example 2. Let us show (part of) the result of applying the points-to analysis to each program point. Since ϵ is the first element of all allocation sequences, we omit it.

`void start (Int n, Int m) {`	$\{this \mapsto \{o_1\}\}$
`  Fut f<void> = this!initDAO();`	$\{this \mapsto \{o_1\}, o_1.dao \mapsto \{o_{13}\}\}$
`  await f?;`	$\{this \mapsto \{o_1\}, o_1.dao \mapsto \{o_{13}\}\}$
`  while(n > 0) {`	$\{this \mapsto \{o_1\}, o_1.dao \mapsto \{o_{13}\}\}$
② `H h = new Handler(this.dao);`	$\{this \mapsto \{o_1\}, o_1.dao \mapsto \{o_{13}\}, h \mapsto \{o_{12}\}\}$
`    h!run(m);`	$\{this \mapsto \{o_1\}, o_1.dao \mapsto \{o_{13}\}, h \mapsto \{o_{12}\}\}$
`    n = n - 1; } }`	$\{this \mapsto \{o_1\}, o_1.dao \mapsto \{o_{13}\}, h \mapsto \{o_{12}\}\}$
`void initDAO () {`	$\{this \mapsto \{o_1\}\}$
③ `this.dao = new DAO(null);`	$\{this \mapsto \{o_1\}, o_1.dao \mapsto \{o_{13}\}\}$
`  Fut f<void> = this.dao!initDB();`	$\{this \mapsto \{o_1\}, o_1.dao \mapsto \{o_{13}\}\}$
`  await f?;`	$\{this \mapsto \{o_1\}, o_1.dao \mapsto \{o_{13}\}\}$
`void initDB () {`	$\{this \mapsto \{o_{13}\}\}$
④ `this.db = new DB();}`	$\{this \mapsto \{o_{13}\}, o_{13}.db \mapsto \{o_{134}\}\}$

All object creations use the object name(s) pointed to by this to generate new object names by adding the current allocation site. E.g., at p.p. ②, $this \mapsto \{o_1\}$; the new object name created is o_{12}. The set of possible values for this within a method comes from the object name(s) for the variable used to call the method. In what follows, we use $\mathcal{O}$ to refer to the set of object names generated by the points-to analysis. In our example $\mathcal{O}=\{o_\epsilon, o_1, o_{12}, o_{13}, o_{134}\}$. □

3.2 Cost Models

Cost models determine the type of resource we are measuring. Traditionally, a cost model $\mathcal{M}$ is a function $\mathcal{M} : Instr \mapsto \mathcal{N}$ which for each instruction in the set of instructions $Instr$ returns a natural number which represents its cost. As an example, if we are interested in counting the number of instructions executed by a program, we define a cost model that counts one unit for any instruction, i.e., $\mathcal{M}(b) = 1$.

In the context of distributed programs, the main difference is that the cost model not only accounts for the cost consumed by the instruction, but it also needs to attribute it to the corresponding cost center. In order to do so, we add an additional parameter to the previous model which corresponds to the allocation site of the cost center: $\mathcal{M}^I(b, o){=}c(o){\cdot}1$. As before, we count "1" instruction but now we attribute it to the cost center of the provided object, named $c(o)$. Technically, the way to assign the cost to its corresponding center is by using symbolic cost expressions that contain the cost centers such that, if we are interested in knowing how many instructions have been executed by the cost center $c(o)$, we replace $c(o)$ by 1 and $c(o')$ by 0 for all other $o'{\neq}o$. In the following sections, we will define the cost models that we need for our analysis.

3.3 Upper Bounds

Given a set of cost centers O, a definition of cost model $\mathcal{M}$, and a program $P(\overline{x})$, where $\overline{x}$ are the input values for the arguments of the main method, resource analysis obtains an *upper bound* $UB_P^{\mathcal{M}}(\overline{x})$ which is an expression of the form $c(o_1){\cdot}e_1 + \ldots + c(o_n){\cdot}e_n$ where $o_i \in O$ and e_i is a cost expression (e.g., polynomials, exponential functions, etc.) with $i = 1, \ldots, n$.

The analysis [1] is object-sensitive in that, given an object x at a program point p, it considers the cost for all different possible abstract values in O that x can take. Technically, this is done by generating cost equations for each possible abstract value and taking the maximum. To allow this object-sensitive extension, the cost model receives the particular allocation site which is being considered by the analysis. The analysis guarantees that $UB_P^{\mathcal{M}}$ is an upper bound on the worst-case cost (for the type of resource defined by $\mathcal{M}$) of the execution of P w.r.t. any input data; and in particular, that each e_i is an upper bound on the execution cost performed within the objects that o_i represents.

Formally, the following theorem states the soundness result of the analysis. Since the length of object names is limited to a length k, allocation sequences of length greater than k do not appear as such in the results of points-to analysis. Instead, they are represented by object names which cover them. Therefore, we need some means for relating allocation sequences to the object name which best approximate them. We now define such notion. Given an allocation sequence l and a set of object names O, the *best approximation* of l in O is the longest object name in O which covers l. I.e., an object name $o_{l'} \in O$ is the best approximation of l in O iff $o_l \le o_{l'}$ and $\forall o_{l''} \in O \,.\, o_l \le o_{l''} \rightarrow |l''| < |l'|$ or $l'' = l'$. We use $UB_P^{\mathcal{M}}(\overline{x})|_N$ to denote the result of replacing $c(o_l)$ by 1 if $o_l \in N$ and by 0 otherwise in the resulting UB expression.

Theorem 1 (soundness [1,3]). *Let P be a program and l an allocation sequence. Let O be the object names computed by a points-to analysis of P. Let l' be the best approximation of l in O. Then, $\forall \overline{x}, cost(l, P, \overline{x}) \le UB_P^{\mathcal{M}}(\overline{x}_s)|_{\{o_{l'}\}}$.*

Example 3. The UB expression obtained by applying resource analysis on the running example using $\mathcal{M}^I$, which counts the number of instructions executed by each object inferred by the points-to analysis, is $UB^{\mathcal{M}^I}_{main}(n,m) = c(o_1)\cdot 18 + c(o_{13})\cdot 6 + c(o_1)\cdot 12\cdot \mathsf{nat}(n) + c(o_{12})\cdot 6\cdot \mathsf{nat}(n) + c(o_{12})\cdot 8\cdot \mathsf{nat}(n)\cdot \mathsf{nat}(m) + c(o_{13})\cdot 2\cdot \mathsf{nat}(n)\cdot \mathsf{nat}(m) + c(o_{134})\cdot \mathsf{nat}(n)\cdot \mathsf{nat}(m)$, where $\mathsf{nat}(x) = max(x,0)$ and it is used for avoiding negative evaluations of cost expressions. In what follows, for readability, nat is omitted from the UB expressions. The number of instructions executed by a particular object name, say o_{12}, is obtained as $UB^{\mathcal{M}^I}_{main}(n,m)|_{\{o_{12}\}} = n\cdot(6+8\cdot m)$. Although the resource analysis in [1,3] cannot infer how object identifiers are grouped in the configuration of the program, it can give us the cost executed by a set of objects, $UB^{\mathcal{M}^I}_{main}(n,m)|_{\{o_1,o_{12}\}} = 18 + n\cdot(18+8\cdot m)$.

4 Concrete Definitions in Distributed Systems

This section formalizes the concrete notions of configuration and communication that we aim at approximating by static analysis in the next section.

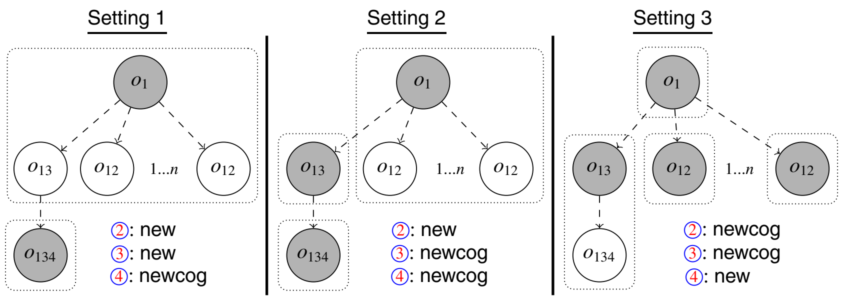
4.1 Configuration

Let us introduce some notation. All instructions are labeled. The expression $b \equiv q : i$ denotes that the instruction b has q as label (program point) and i is the instruction proper. Similarly to the points-to analysis defined in Sec. 3.1, any object can be assigned an *allocation sequence* $l = \langle j, \ldots, p, q\rangle$ which indicates that such object was allocated at program point q during the execution of a method invoked on an object whose allocation sequence is in turn $\langle j, \ldots, p\rangle$. We use o_l to refer to an object whose allocation sequence is l. Note that allocation sequences are not identifiers since there may be multiple objects with the same allocation sequence. Therefore, we sometimes use multisets (denoted $\{\!\!|\ |\!\!\}$). Underscores (_) are used to ignore irrelevant information. Given an allocation sequence l, we use $root(l)$ to refer to the allocation sequence of the root object of the cobox for l. It can be defined as the longest prefix of l which ends in an allocation site for coboxes, i.e., one site where a newcog instruction is executed. Therefore, if l ends in an allocation site for coboxes, then $root(l) = l$. If it ends in an allocation site for objects, i.e., one where a new instruction is executed, then $root(\langle j, \ldots, p, q\rangle) = root(\langle j, \ldots, p\rangle)$.

Given a trace t (see Section 2), we use $steps(t)$ to denote the set of steps which form trace t. Since execution is non-deterministic, given a program $P(\overline{x})$, multiple (possibly infinite) fully expanded traces may exist. We use $executions(P(\overline{x}))$ to denote the set of all possible fully expanded traces for $P(\overline{x})$. Given a trace t, the multiset of cobox roots created during t is defined as $cobox\_roots(t) = \{\!\!|o_l \mid \_ \leadsto^{q:\mathsf{newcog}\ \_}_{\langle j,\ldots,p\rangle}\ \_ \in steps(t) \wedge\ l = \langle j, \ldots, p, q\rangle|\!\!\}$. Also, given a cobox root o_l, the multiset of objects it owns in a trace t is defined as $abs\_in\_cobox(o_l, t) = \{\!\!|o_{\langle j,\ldots,p,q\rangle} | \_ \leadsto^{q:\mathsf{new}\ \_}_{\langle j,\ldots,p\rangle}\ \_ \in steps(t) \wedge root(\langle j, \ldots, p\rangle) = l|\!\!\}$.

Definition 1 (configuration). *Given an execution trace t, we define its* configuration*, denoted* C_t*, as* $C_t = \{\!\!|\langle o, abs\_in\_cobox(o,t)\rangle \mid o \in cobox\_roots(t)|\!\!\}$*. The* configuration *of a program P on input values* $\overline{x}$*, denoted* $Conf_P(\overline{x})$ *is defined as* $\{C_t \mid t \in executions(P(\overline{x}))\}$.

Example 4. Deliberately, the running example shown in Ex. 1 executes in a single cobox. It can be configured as a distributed application by creating coboxes instead of objects, i.e., by replacing selected new instructions by newcog . The following graphs graphically show three possible settings and the memory allocation instruction (new or newcog) that have been used at the program points ②, ③ and ④.



The object names in the graph are grouped using dotted rectangles according to the cobox to which they belong. Cobox roots appear in grey and dashed edges represent the creation sequence. The annotation $1 \dots n$ indicates that we have n objects of this form. In *Setting 1*, all objects are created in the same cobox, except for the object of type DB. In *Setting 2*, also the object of type DAO is in a separate cobox. In *Setting 3*, each handler is in a separate cobox, and DAO and DB share a cobox. The configurations for the different settings are (see Def. 1):

$$\textit{Setting 1:} \{\!|\langle o_1, \{\!| \underbrace{o_{12}, ..., o_{12}}_{n \text{ objects}}, o_{13} |\!\}\rangle, \langle o_{134}, \{\!||\!\}\rangle |\!\} \qquad \textit{Setting 2:} \{\!|\langle o_1, \{\!| \underbrace{o_{12}, ..., o_{12}}_{n \text{ objects}} |\!\}\rangle, \langle o_{13}, \{\!||\!\}\rangle, \langle o_{134}, \{\!||\!\}\rangle |\!\}$$

$$\textit{Setting 3:} \{\!|\langle o_1, \{\!||\!\}\rangle, \underbrace{\langle o_{12}, \{\!||\!\}\rangle, ..., \langle o_{12}, \{\!||\!\}\rangle}_{n \text{ coboxes}}, \langle o_{13}, \{\!| o_{134} |\!\}\rangle |\!\}$$

□

Given input values and an allocation sequence, the definition below counts the maximum number of instances (objects) created at such allocation sequence. We use $card(x, M)$ to refer to the number of occurrences of x in a multiset M.

Definition 2 (number of instances). *Given an allocation sequence l, a program P and input values $\overline{x}$, we define the* number of instances *for l as:*

$$inst(l, P, \overline{x}) = \max_{C_t \in Conf_P(\overline{x})} \Big(\sum_{\langle c,O \rangle \in C_t} card(l, O \cup \{\!| c |\!\}) \Big)$$

Example 5. The number of instances for the allocation sequence $\langle 1, 2 \rangle$ in our running example with input values $\overline{x} = \langle 3, 4 \rangle$ (i.e., n=3 and m=4) is the maximum number of objects with $\langle 1, 2 \rangle$ as allocation sequence, over all possible executions. Such maximum is 3. In fact, for any execution the maximum coincides with the value of n. □

4.2 Communication

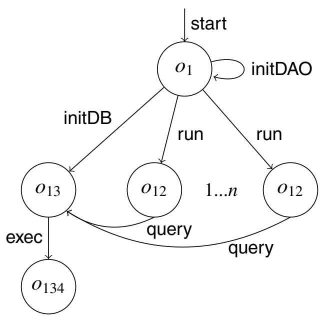
The *communication* refers to the *interactions* between objects occurred during the execution of a program. As in the above section, objects are represented using allocation sequences.

Definition 3 (communication). *Given an execution trace t, its* interactions*, denoted I_t, are defined as: $I_t = \{\!|\langle o_l, o_k, m\rangle \mid \_ \leadsto_l^{\_:\,o_k!m(\_)} \_ \in steps(t)|\!\}$. The* communication *performed in the execution of a program P and input values $\overline{x}$, denoted $Comm_P(\overline{x})$ is defined as $\{I_t \mid t \in executions(P(\overline{x}))\}$.*

A global view of the distributed system for a trace execution t can be depicted as a graph whose nodes are object representations of the form o_l, where l is an allocation sequence which occurs in the trace t, and whose arcs, annotated with the method name, are given by the elements in the set I_t.

Example 6. The interactions for any execution of our running example, and thus, the communication of the program, is depicted graphically in the following graph and, according to Def. 3 it is defined as:

$$\{\!|\langle \epsilon, o_1, \mathsf{start}\rangle, \langle o_1, o_1, \mathsf{initDAO}\rangle, \langle o_1, o_{13}, \mathsf{initDB}\rangle, \underbrace{\langle o_1, o_{12}, \mathsf{run}\rangle, ..., \langle o_1, o_{12}, \mathsf{run}\rangle,}_{n \text{ interactions}} \underbrace{\langle o_{12}, o_{13}, \mathsf{query}\rangle, ..., \langle o_{12}, o_{13}, \mathsf{query}\rangle}_{n\cdot m \text{ interactions}} \underbrace{\langle o_{13}, o_{134}, \mathsf{exec}\rangle, ..., \langle o_{13}, o_{134}, \mathsf{exec}\rangle}_{n\cdot m \text{ interactions}} |\!\}$$



Observe that the communication of the program comprises all calls to methods, including calls within the same object such as $\langle o_1, o_1, \mathsf{initDAO}\rangle$. A relevant aspect of communications is that they are independent from the distributed setting of the program. □

Definition 4 (number of interactions). *Given two allocation sequences l and k, a method m, a program P and its input values $\overline{x}$, we define the* number of interactions *between l and k for method m in the execution of P on $\overline{x}$ as: $ninter(l, k, m, P, \overline{x}) = \max_{I_t \in Comm_P(\overline{x})} (card(\langle l, k, m\rangle, I_t))$.*

Example 7. In the running example, methods initDAO and initDB are executed only once. During the execution of start in object o_1, method run is called inside the while loop and it is executed n times by the objects o_{12}. Similarly, for each execution of run in o_{12}, method query is called m times, resulting in $n \cdot m$ calls to method query in o_{13}. Besides, each call to query executes exec in o_{134}. □

5 Inference of Quantified Abstractions

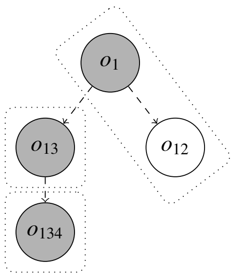
This section presents our method to infer quantified abstractions of distributed systems. The main novelties are: (1) We provide an abstract definition for configuration and communication that can be automatically inferred by relying on the results computed by points-to analysis. (2) We enrich the abstraction by integrating quantitative information inferred by resource analysis. For this, we build on prior work on resource analysis [1,3] that was primarily used for the estimation of upper bounds on the worst-case cost performed by each node in the system (see Section 3). To use this analysis, we need to define new cost models that allow establishing upper bounds for the number of nodes and communications which the execution of the system requires.

5.1 Quantified Configurations

The points-to analysis results can be presented by means of a *points-to graph* as follows. We use *alloc*(P) to denote the set of allocation sites in program P.

Definition 5 (points-to graph). *Given a program P and its points-to analysis results, we define its* points-to graph *as a directed graph* $G_P = \langle V, E \rangle$ *whose set of nodes is* $V = \hat{O}$ *and set of edges is* $E = \{o_l \rightarrow o_{l'} \mid q{:}y{=}\mathsf{new}\ \_ \textit{ or } q{:}y{=}\mathsf{newcog}\ \_ \in alloc(P) \wedge o_l \in pt(q, this) \wedge o_{l'} \in pt(q, y)\}$.

Example 8. The following graph shows the points-to graph for the running example. It contains one node for each object name inferred by the points-to analysis. Given an allocation site, edges link object names pointed to by this to the corresponding objects created at that program point, e.g., an edge from o_1 to o_{13} and o_{12} and another one from o_{13} to o_{134}. □



Points-to graphs provide abstractions of the ownership relations among objects in the program. To extract abstract configurations from them, it is necessary to identify cobox roots and find the set of objects which belong to the coboxes associated to such roots. Note that given an object name $\langle j \ldots q \rangle$ it can be decided whether it represents a cobox root, denoted *is_root*$(\langle j \ldots q \rangle)$, by simply checking whether the allocation site q contains a newcog instruction. We write $a \rightsquigarrow b$ to indicate that there is a non-empty path in a graph from a to b and denote by *interm*(a, b) the set of intermediate nodes in the path (excluding a and b).

Definition 6 (abstract configuration). *Given a program P and a points-to graph* $G_P = \langle V, E \rangle$ *for P, we define its* abstract configuration $\mathcal{A}_P$ *as the set of pairs of the form* $\langle o, abs\_in\_cobox(o, G_P) \rangle$ *s.t.* $o \in V \wedge is\_root(o)$ *where* $abs\_in\_cobox(o, G_P) = \{o' \in V$ *s.t.* $o \rightsquigarrow o'$ *in* G_P *and* $\forall o'' \in interm(o, o') \ \wedge \ \neg is\_root(o'')\}$.

Note that, in the above definition, function *abs_in_cobox* returns the subset of objects which are part of the cobox whose root is the parameter o.

Example 9 (abstract configuration). The abstract configuration for the concrete *Setting 2* is represented graphically in Ex. 8. As before, cobox roots appear in grey and objects are grouped by cobox. The abstract configurations for the Settings in Ex. 4 are:
Setting 1: $\langle o_1, \{o_{12}, o_{13}\} \rangle, \langle o_{134}, \{\} \rangle$, *Setting 2*: $\langle o_1, \{o_{12}\} \rangle, \langle o_{13}, \{\} \rangle, \langle o_{134}, \{\} \rangle$,
Setting 3: $\langle o_1, \{\} \rangle, \langle o_{12}, \{\} \rangle, \langle o_{13}, \{o_{134}\} \rangle$ □

Soundness of the analysis requires that the abstract configuration obtained is a safe approximation of the configuration of the program for any input values. Given two object names o_l and $o_{l'}$, we say that $o_{l'}$ *covers* o_l, written $o_l \preceq o_{l'}$ if l' is a suffix of l modulo $\oplus$. Given two sets of object names O_1 and O_2, we write $O_1 \sqsubseteq O_2$ if all objects in O_1 are covered by some element in O_2. Given $\langle o_l, O \rangle$ and $\langle o_{l'}, O' \rangle$, we write $\langle o_l, O \rangle \sqsubseteq \langle o_{l'}, O' \rangle$ if $o_l \preceq o_{l'}$ and $O \sqsubseteq O'$. Given two configurations C and C', we write $C \sqsubseteq C'$ if $\forall \langle o_l, O \rangle \in C$ there exists $\langle o_{l'}, O' \rangle \in C'$ s.t. $\langle o_l, O \rangle \sqsubseteq \langle o_{l'}, O' \rangle$.

Theorem 2 (soundness of abstract configurations). *Let P be a program and $\mathcal{A}_p$ its abstract configuration. Then $\forall \overline{x}, \forall C_t \in Conf_P(\overline{x}), C_t \sqsubseteq \mathcal{A}_p$.*

The proof is entailed from the soundness proof of the underlying points-to analysis (our implementation uses an adaptation of [11]). It is easy to see that the theorem holds for the configuration $Conf_P$ in Ex. 4, and any abstract configuration $\mathcal{A}_P$ of Ex. 9.

(Non-quantified) abstract configurations are already useful when combined with the resource analysis in Sec. 3, since they allow us to obtain the resource consumption at the level of cobox names. In what follows, given a points-to graph G_P and a cobox root o, we use $cobox(o, G_P,)$ to denote $\{o\} \cup abs\_in\_cobox(o, G_P)$.

Example 10. Using the UB expression inferred in Ex. 3 and the abstract configurations for all settings in Ex. 9, we can obtain the cost for each cobox name. The following table shows the results obtained from $UB^{\mathcal{M}^I}_{main}(n,m)|_{cobox(c,G_{main})}$ where c corresponds, in each case, to the cobox name in the considered abstract configuration:

Setting 1		*Setting 2*		*Setting 3*	
c	UB	c	UB	c	UB
o_1	$24 + 18{\cdot}n + 10{\cdot}n{\cdot}m$	o_1	$18 + 18{\cdot}n + 8{\cdot}n{\cdot}m$	o_1	$18 + 12{\cdot}n$
o_{134}	$n{\cdot}m$	o_{13}	$6 + 2{\cdot}n{\cdot}m$	o_{12}	$6{\cdot}n + 8{\cdot}n{\cdot}m$
		o_{134}	$n{\cdot}m$	o_{13}	$6 + 3{\cdot}n{\cdot}m$

As the table shows, in *Settings 1* and *2* most of the instructions are executed in cobox(es) represented by cobox name o_1. In *Setting 3*, the cost is more evenly distributed among cobox names. However, in order to reason about how loaded actual coboxes are it is required to have information about how many instances of each cobox name exist. For example, in *Setting 3*, o_{12} represents n Handler coboxes. This essential (and complementary) information will be provided by the quantified abstraction. □

We now aim at quantifying abstract configurations, i.e., at inferring an over-approximation of the number of concrete objects (and coboxes) that each abstract object (or cobox) represents. For this purpose, we define the $\mathcal{M}^C(b, o_l)$ cost model as a function which returns $c(o_{l \oplus q})$ if $b \equiv q{:}y{=}\texttt{new}\ C$ or $b \equiv q{:}y{=}\texttt{newcog}\ C$, and 0 otherwise. The novelty is on how the information computed by the points-to analysis is used in the cost model: it concatenates the allocation sequence of the object received as parameter (that corresponds to the considered allocation sequence for this) with the instruction allocation site q. This allows counting the elements created at this point for each particular instance of this considered by the points-to analysis.

Example 11. Using $\mathcal{M}^C$, the upper bound obtained for the running example is the expression $UB^{\mathcal{M}^C}_{main}(n,m) = c(o_1) + c(o_{13}) + c(o_{134}) + n \cdot c(o_{12})$. This expression allows us to infer an upper bound of the maximum number of instances for any object identified in the points-to graph. Regarding configurations, we are interested in the number of instances of those objects that are distributed nodes (coboxes). The following table shows the results of solving the expression $UB^{\mathcal{M}^C}_{main}(n,m)|_{cobox(c,G_{main})}$ where c as before are the coboxes for each abstract configuration.

Setting 1		*Setting 2*		*Setting 3*	
c	*UB*	*c*	*UB*	*c*	*UB*
o_1	1	o_1	1	o_1	1
o_{134}	1	o_{13}	1	o_{12}	n
		o_{134}	1	o_{13}	1
2		3		$2 + n$	

Clearly, *Setting 1* is the setting that creates fewer coboxes (only 2 coboxes execute the whole program). Thus, the queries requested by handlers cannot be processed in parallel. If there is more parallel capacity available, *Setting 3* may be more appropriate, since handlers can process requests in parallel.

Theorem 3. *Under the assumptions in Th. 1,* $\forall\overline{x}$, $inst(l, P, \overline{x}) \leq UB_P^{\mathcal{M}^C}(\overline{x}_s)|_{\{o_{l'}\}}$.

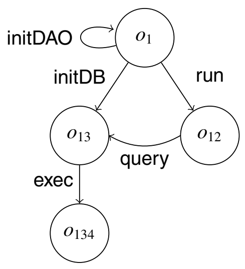
The proof is an instance of Th. 1 for $\mathcal{M}^C$ and the definition of *inst* in Def. 2.

5.2 Quantified Communication

From the points-to analysis results, we can generate the *interaction graph* as follows.

Definition 7 (interaction graph). *Given a program P and its points-to analysis results, we define its* interaction graph *as a directed graph* $I_P = \langle V, E\rangle$ *with a set of nodes* $V = O$ *and a set of edges* $E = \{o_l \xrightarrow{m} o_{l'} \mid q{:}x!m(\_) \wedge o_l \in pt(q, this) \wedge o_{l'} \in pt(q, x)\}$.

Example 12. The following graph shows the interaction graph for the example. Edges connect the object that is executing when a method is called with the object responsible for executing the call, e.g., during the execution of start, object o_1 calls method initDAO using the this reference and it also interacts with o_{12} by calling run. Note that the multiple calls to query from o_{12} to o_{13} are abstracted by one edge. □



We now integrate quantitative information in the interaction graph. For this purpose, we define the cost model $\mathcal{M}^K(b, o, p)$ as a function which returns $c(m){\cdot}c(o, p)$ if $b \equiv \_!m(\_)$, and 0 otherwise. The key point is that for capturing interactions between objects, when applying the cost model to an instruction, we pass as parameters the considered allocation sequences of the caller and callee objects. The resulting upper bounds will contain cost centers made up of pairs of abstractions $c(o, p)$, where o is the object that is executing and p is the object responsible for executing the call. Besides, we attach to the interaction the name of the invoked method $c(m)$ (multiplication is used as an instrument to attach this information and manipulate it afterwards as we describe below).

From the upper bounds on the interactions, we can obtain a range of useful information: (1) By replacing $c(m)$ by 1, we obtain an upper bound on the number of interactions between each pair of objects. (2) We can replace $c(m)$ by (an estimation of) the amount of data transferred when invoking m (i.e., the size of its arguments). This is a first approximation of a band-width analysis. (3) Replacing $c(o, p)$ by 1 for selected objects and the remaining ones by 0, we can see the interactions between the selected objects. (4) If we are interested in the communications for the whole program, we just replace all expressions $c(o, p)$ by 1. (5) Furthermore, we can obtain the interactions between the distributed nodes by replacing by 1 those cost centers in which o and p belong to different coboxes and by 0 the remaining ones. From this information, we can detect nodes that have many interactions and that would benefit from being deployed on the same machine or at least have a fast communication channel.

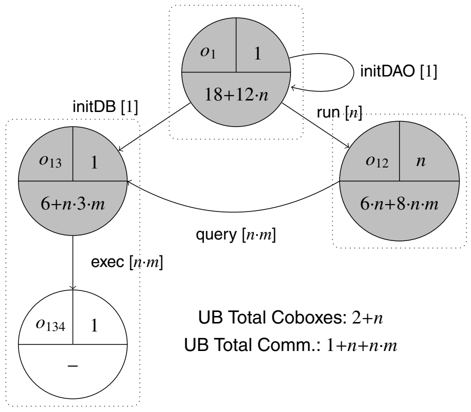
Example 13. The interaction UB obtained by the resource analysis is as follows:

$$UB^{\mathcal{M}^K}_{main}(n,m) = c(\mathsf{start}){\cdot}c(\epsilon,o_1) + c(\mathsf{initDAO}){\cdot}c(o_1,o_1) + c(\mathsf{initDB}){\cdot}c(o_1,o_{13}) + n{\cdot}c(\mathsf{run}){\cdot}c(o_1,o_{12}) + n{\cdot}m{\cdot}c(\mathsf{query}){\cdot}c(o_{12},o_{13}) + n{\cdot}m{\cdot}c(\mathsf{exec}){\cdot}c(o_{13},o_{134})$$

From this global UB, we obtain the following UBs on the number of interactions between coboxes for the different settings in Ex. 4:

Setting 1			*Setting 2*			*Setting 3*		
method	*coboxes*	*UB*	*method*	*coboxes*	*UB*	*method*	*coboxes*	*UB*
exec	$o_1 \to o_{134}$	$n{\cdot}m$	query	$o_1 \to o_{13}$	$n{\cdot}m$	run	$o_1 \to o_{12}$	n
			exec	$o_{13} \to o_{134}$	$n{\cdot}m$	initDB	$o_1 \to o_{13}$	1
						query	$o_{12} \to o_{13}$	$n{\cdot}m$
$n{\cdot}m$			$n{\cdot}m + n{\cdot}m$			$1 + n + n{\cdot}m$		

The last row shows the total number of interactions between coboxes. Clearly, the minimum number of inter-cobox interactions happens in *Setting 1*, where most of the objects are in the same cobox. *Setting 2* has a higher number of interactions, because the database objects DAO and DB are in different coboxes. In *Setting 3* most interactions are produced between the coboxes created for the handlers which, on the positive side, may run in parallel. By combining this information with the quantified configuration of the system, for *setting 3*, we generate the *quantified abstraction* (shown in the graph). Each node contains as object identifier its allocation sequence and the number of instances (e.g., the number of instances of o_{12} is n). Optionally, if it is a cobox, it contains the number of instructions executed by it. For instance, the UB on the number of instructions executed in cobox o_{12} is $6{\cdot}n{+}8{\cdot}n{\cdot}m$ (see Ex. 10). The edges represent the interactions and are annotated (in brackets) with the UB on the number of calls (e.g., the objects represented by o_{12} call to o_{13} $n{\cdot}m$ times calling method query). □



From the example, we can figure out the five applications described in Sec. 1: (1) We can visualize the topology and view the number of tasks to be executed by the distributed nodes and possibly spot errors. (2) We detect that node o_1 executes only one process, while o_{13} executes many. Thus, it probably makes sense to have them sharing the processor. (3) We can perform meaningful resource analysis by assigning to each distributed node the number of steps performed by it, rather than giving this number at the level of objects as in [1,3] (as maybe the objects do not share the processor). (4) We can see that o_{13} and o_{12} have many interactions and would benefit from having a fast communication channel. (5) From the quantified interactions, if we compute the sizes of the arguments, we can figure out the size of the data to be transferred (bandwidth

Similarly to abstract configurations, we use $UB^{\mathcal{M}^K}_{main}(\overline{x})|_{N,M}$ to denote the result of replacing in the resulting UB the expression: $c(o_1, o_2)$ by 1 if $o_1, o_2 \in N$ and by 0 otherwise and $c(m)$ by 1 if $m \in M$ and by 0 otherwise. This theorem is also an instance of Th. 1 for the defined cost model and Def. 4.

Theorem 4 (soundness). *Under the assumptions in Theorem 1, $\forall\ \overline{x}$ we have that $ninter(l, k, m, P, \overline{x}) \leq UB^{\mathcal{M}^K}_{P}(\overline{x})|_{\{o_l,o_k\},\{m\}}$.*

6 Implementation and Application to Case-Study

We have implemented our analysis in COSTABS [1] and applied it to a realistic case-study, the Trading System developed by Fredhopper® and available from `http://www.hats-project.eu`. Due to some limitations of the underlying resource analysis which are not related to our method, we had to slightly modify the program by changing the structure of some loops, and had to add some size relations that the analysis could not infer. The simple online interface to our analysis and the modified case-study can be found at `http://costa.ls.fi.upm.es/costabs/QA`. This Trading System is a typical example for a distributed component-based information system. It models a supermarket sales handling system: it includes the processes at a single cash desk (like scanning products using a bar code scanner or paying by cash or by credit card); it also handles bill printing; as well as administrative tasks In the Trading System, a store consists of an arbitrary number of cash desks. Each of them is connected to the store server, holding store-local product data such as inventory stock, prices, etc.

The experiments presented have been performed on an Intel Core 2 Duo at 2.53GHz with 4GB of RAM, running Ubuntu 12.10. The analyzed source code has 1340 l.o.c., with 94 methods and 22 classes. Points-to analysis has been performed with $k = 2$, i.e., the maximum length of object names (see Sec. 3) is two. The inference of the quantified configuration took 109 seconds and of the quantified communication 410 seconds. The larger time taken by the communication is justified because there are many more method invocations than object creations in the program and, thus, the resource analysis has more equations to solve in the latter case. The analysis identifies 22 different object names (17 of them are coboxes) and 96 interactions in the communication graph. We do not show the UBs inferred because the expressions obtained are rather large.

The QA obtained for the system is as follows: we identify two separate parts in the model, an environment part that creates a handler for each physical device and another part that represents the physical devices. The configuration of the system is distributed by creating one cobox for each physical device. The quantified abstraction infers that the number of instances for each identified cobox is linear on the number of cash desks installed. Besides, the system also creates one distributed environment object running on its own cobox for handling each physical device. The interactions of the system show that each environment cobox communicates with its physical device in order to perform each task. A notable result of our experiments is that we have detected two objects with a high number of interactions, namely *CashDeskPCImpl* and *CashBoxImpl*, and which run in separate coboxes. Clearly, the implementation would benefit from deploying these two coboxes on the same machine since their tasks are highly cooperative. If this is not possible, it should be at least guaranteed that they have a fast communication channel. The remaining objects do not show any overloading problem.

7 Conclusions and Future Work

We have shown that distributed systems can be statically approximated, both qualitatively and quantitatively. For this, we have proposed the use of powerful techniques for points-to and resource analysis whose integration results in a novel approach to describing system configurations. There exist several contributions in the literature about occurrence counting analysis in mobile systems of processes, although they focus on high-level models, such as the π-calculus and BioAmbients [8,9]. But, to the best of our knowledge, this paper is the first approach that presents a quantitative abstraction of a distributed system for a real language and experimentally evaluates on a prototype. We argue that our work is a first crucial step towards automatically inferring optimal deployment configurations of distributed systems. In future work, we plan to tackle this problem and consider objective functions. An objective function should indicate the cost metrics that we aim at keeping minimal, e.g., by taking into account the actual features of the deployment platforms.

Acknowledgments. This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, and by the Spanish projects TIN2008-05624, TIN2012-38137, PRI-AIBDE-2011-0900 and S2009TIC-1465 *PROMETIDOS-CM*.

References

1. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: Cost Analysis of Concurrent OO Programs. In: Yang, H. (ed.) APLAS 2011. LNCS, vol. 7078, pp. 238–254. Springer, Heidelberg (2011)
2. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: COSTABS: A Cost and Termination Analyzer for ABS. In: Procs. of PEPM 2012, pp. 151–154. ACM Press (2012)
3. Albert, E., Arenas, P., Correas, J., Gómez-Zamalloa, M., Genaim, S., Puebla, G., Román-Díez, G.: Object-sensitive cost analysis for concurrect objects. Technical Report (2012), `http://costa.ls.fi.upm.es/papers/costa/AlbertACGGPRtr.pdf`
4. America, P.: Issues in the design of a parallel object-oriented language. Formal Aspects of Computing 1, 366–411 (1989)
5. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Computer Systems 25(6), 599–616 (2009)
6. Caromel, D.: Towards a method of object-oriented concurrent programming. Communications of the ACM 36(9), 90–102 (1993)
7. Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Restraints Among Variables of a Program. In: POPL, ACM Press (1978)
8. Feret, J.: Occurrence counting analysis for the pi-calculus. ENTCS 39(2), 1–18 (2001)

9. Gori, R., Levi, F.: A new occurrence counting analysis for bioambients. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 381–400. Springer, Heidelberg (2005)
10. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A Core Language for Abstract Behavioral Specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
11. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for java. ACM Trans. Softw. Eng. Methodol. 14, 1–41 (2005)
12. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing Active Objects to Concurrent Components. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 275–299. Springer, Heidelberg (2010)
13. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: understanding object-sensitivity. In: Procs. of POPL 2011, pp. 17–30. ACM (2011)
14. Yonezawa, A., Briot, J.P., Shibayama, E.: Object-oriented concurrent programming ABCL/1. In: Procs. of OOPLSA 1986, pp. 258–268. ACM, USA (1986)

An Algebraic Theory for Web Service Contracts

Cosimo Laneve[1] and Luca Padovani[2]

[1] Università di Bologna – INRIA Focus Team, Italy
[2] Università di Torino – Dipartimento di Informatica, Italy

Abstract. We study a natural notion of compliance between clients and services in terms of their BPEL (abstract) descriptions. The induced preorder shows interesting connections with the *must* preorder and has normal form representatives that are parallel-free finite-state activities, called *contracts*. The preorder also admits the notion of least service contract that is compliant with a client contract, called *principal dual contract*. Our framework serves as a foundation of Web service technologies for connecting abstract and concrete service definitions and for service discovery.

1 Introduction

Service-oriented technologies and Web services have been proposed as a new way of distributing and organizing complex applications across the Internet. These technologies are nowadays extensively used for delivering cloud computing platforms.

A large effort in the development of Web services has been devoted to their specification, their publication, and their use. In this context, the Business Process Execution Language for Web Services (BPEL for short) has emerged as the de-facto standard for implementing Web service composition and, for this reason, it is supported by the toolkits of the main software vendors (Oracle Process Manager, IBM WebSphere, and Microsoft BizTalk).

As regards publication, service descriptions should retain abstract (behavioral) definitions, which are separate from the binding to a concrete protocol or endpoint. The current standard is defined by the Web Service Description Language (WSDL) [10], which specifies the format of the exchanged messages – the *schema* –, the locations where the interactions are going to occur – the *interface* –, the transfer mechanism to be used (i.e. SOAP-RPC, or others), and basic service abstractions (one-way/asynchronous and request-response/synchronous patterns of conversations). Since these abstractions are too simple for expressing arbitrary, possibly cyclic protocols of exchanged messages between communicating parties, WSDL is not adequate to verify the behavioral compliance between parties. It is also worth to notice that the attempts, such as UDDI (Universal Description, Discovery and Integration) registries [5], provide limited support because registry items only include pointers to the locations of the service abstractions, without constraining the way these abstractions are defined or related to the actual implementations (*cf.* the `<tModel>` element). In this respect, UDDI registries are almost useless for discovering services; an operation that is performed manually by service users and consumers.

The publication of abstract service descriptions, called *contracts* in the following, and the related ability of service searching assume the existence of a formal notion of contract equivalence and, more generally, of a formal theory for reasoning about Web services by means of their contracts. We identify three main goals of a theory of Web service contracts: (1) it should provide a formal language for describing Web services at a reasonable level of abstraction and for admitting static correctness verification of client/service protocol implementations; (2) it should provide a semantic notion of contract equivalence embodying the principle of safe Web service replacement. Indeed, the lack of a formal characterization of contracts only permits excessively demanding notions of equivalence such as nominal or structural equality; (3) it should provide tools for effectively and efficiently searching Web services in Web service repositories according to their contract.

The aim of this contribution is to provide a suitable theory of contracts for Web services by developing a semantic notion of contract equivalence. In fact, we relax the equivalence into a *subcontract preorder*, so that Web services exposing "larger" contracts can be *safely* returned as results of queries for Web services with "smaller" contracts. We will precisely define what "smaller" and "larger" mean, and we will define which safety property we wish to preserve when substituting a service exposing a contract with a service exposing a larger contract. Our investigation abstracts away from the syntactical details of schemas as well as from those aspects that are oriented to the actual implementations, such as the definition of transmission protocols; all these aspects may be easily integrated on top of the formalism. We do not commit to a particular interpretation of the actions occurring in contracts either: they can represent different typed channels on which interaction occurs or different types of messages.

To equip contracts with a subcontract preorder, we commit to a testing approach. We define client satisfaction as the ability of the client to successfully complete *every* interaction with the service; here "successfully" means that the client never gets stuck (this notion is purposefully asymmetric as client's satisfaction is our main concern). The preorder arises by comparing the sets of clients satisfied by services.

The properties enjoyed by the subcontract preorder are particularly relevant in the context of Web services. Specifically, it is possible to determine, given a client exposing a certain behavior, the smallest (according to subcontract preoder) service contract that satisfies the client – the *principal dual contract*. This contract, acting like a *principal type* in type systems, guarantees that a query to a Web service registry is answered with the largest possible set of compatible services in the registry's databases.

Related Works. Our contracts are normal forms of τ-less CCS processes, a calculus developed by De Nicola and Hennessy in a number of contributions [13,15,18]. The use of formal models to describe communication protocols is not new (see for instance the exchange patterns in SSDL [20], which are based on CSP and the

π-calculus), nor is it the use or CCS processes as behavioral types (see [19] and [9]). The subcontract relation $\lesssim$ has been introduced in [17]. In [6] the authors have studied a refined version of $\lesssim$ that is more suited for orchestrations. The works that are more closely related to ours are by Castagna *et al.* [8] and the ones on *session types*, especially [14] by Gay and Hole. The authors of [8] make the assumption that client and service can be mediated by a *filter*, which prevents potentially dangerous interactions by dynamically changing the interface of the service as it is seen by the client. The present work can be seen as a special case of [8] in which the filter is static and consequently is unnecessary; at the same time, in the present work we also consider divergence, which is not addressed in [8]. With respect to [14] (and systems based on session types) our contract language is much simpler and it can express more general forms of interaction. While the language defined in [14] supports first-class sessions and name passing, it is purposefully tailored so that the transitivity problems mentioned above are directly avoided at the language level. This restricts the subcontract relation in such a way that internal and external choices can never be related (hence, $\{a, b\} : a \oplus b \preceq \{a, b\} : a + b$ does *not* hold).

As regards schemas, which are currently part of BPEL contracts, it is worth mentioning that they have been the subject of formal investigation by several research projects [4, 16]. This work aims at pursuing a similar objective, but moving from the description of data to the description of behaviors.

Structure of the Paper. In Section 2 we introduce BPEL abstract activities and their semantics. In Section 3 we define contracts and detail their relationship with BPEL abstract activities. In Section 4 we address the issue of service discovery in repositories. We conclude in Section 5. Due to space limitations, proofs have been omitted; they can be found in the full paper.

2 BPEL and the Abstract Language

We introduce the basic notions of BPEL by means of an example. The XML document in Figure 1 describes the behavior of an order service that interacts with four other partners, one of them being the customer (`purchasing`), the others being providers of price (`invoicing` service), shipment (`shipping` service), and manufacturing scheduling (`scheduling` service). The business process is made of *activities*, which can be either *atomic* or *composite*. In this example atomic activities are represented by *invocation* of operations in other partners (lines 7–9, 15–18, 22–25), *acceptance* of messages from other partners, either as incoming requests (line 3) or as responses to previous invocations (lines 10–12 and 19), and *sending* of responses to clients (line 28). Atomic activities are composed together into so-called structured activities, such as *sequential composition* (see the `sequence` fragments) and *parallel composition* (see the `flow` fragment at lines 4–27). In a `sequence` fragment, all the child activities are executed in the order in which they appear, and each activity begins the execution only after the previous one has completed. In a `flow` fragment, all the

```
 1 <process>
 2   <sequence>
 3     <receive partnerLink="purchasing" operation="sendPurchaseOrder"/>
 4     <flow>
 5       <links> <link name="ship-to-invoice"/> <link name="ship-to-scheduling"/> </links>
 6       <sequence>
 7         <invoke partnerLink="shipping" operation="requestShipping">
 8           <sources> <source linkName="ship-to-invoice"/> </sources>
 9         </invoke>
10         <receive partnerLink="shipping" operation="sendSchedule">
11           <sources> <source linkName="ship-to-scheduling"/> </sources>
12         </receive>
13       </sequence>
14       <sequence>
15         <invoke partnerLink="invoicing" operation="initiatePriceCalculation"/>
16         <invoke partnerLink="invoicing" operation="sendShippingPrice">
17           <targets> <target linkName="ship-to-invoice"/> </targets>
18         </invoke>
19         <receive partnerLink="invoicing" operation="sendInvoice"/>
20       </sequence>
21       <sequence>
22         <invoke partnerLink="scheduling" operation="requestProductionScheduling"/>
23         <invoke partnerLink="scheduling" operation="sendShippingSchedule">
24           <targets> <target linkName="ship-to-scheduling"/> </targets>
25         </invoke>
26       </sequence>
27     </flow>
28     <reply partnerLink="purchasing" operation="sendPurchaseOrder"/>
29   </sequence>
30 </process>
```

Fig. 1. BPEL business process for an e-commerce service

child activities are executed in parallel, and the whole `flow` activity completes as soon as *all* the child activities have completed. It is possible to constrain the execution of parallel activities by means of *links*. In the example, there is a link `ship-to-invoice` declared at line 5 and used in lines 8 and 17, meaning that the invocation at lines 16–18 cannot take place before the one at lines 7–9. Similarly, the link `ship-to-scheduling` means that the invocation at lines 23–25 cannot take place before the receive operation at lines 10–12 has completed. Intuitively, the presence of links limits the possible interleaving of the activities in a `flow` fragment.

Note that BPEL includes other conventional constructs not shown in the example, such as conditional and iterative execution of activities.

To pursue our formal investigation, we will now present an abstract language of processes whose operators correspond to those found in BPEL. Since we will focus on the interactions of BPEL activities with the external environment, rather than on the actual implementation of business processes, our process language overlooks details regarding internal, unobservable evaluations. For example, the BPEL activity

```
<if>
  <condition> bool-expr </condition>
  activity-True
  <else> activity-False </else>
</if>
```

will be abstracted into the process `activity-True` $\oplus$ `activity-False`, meaning that one of the two activities will be performed and the choice will be a consequence of some unspecified internal decision. A similar observation pertains to the `<while>` activity (see Remark 2).

2.1 Syntax of BPEL Abstract Activities

Let N be a set of *names*, ranged over by $a, b, c, \ldots$, and $\overline{\mathsf{N}}$ be a disjoint set of *co-names*, ranged over by $\overline{a}, \overline{b}, \overline{c}, \ldots$; the term *action* refers to names and co-names without distinction; actions are ranged over by $\alpha, \beta, \ldots$. Let $\overline{\overline{a}} = a$. We use $\varphi, \psi, \ldots$ to range over $(\mathsf{N} \cup \overline{\mathsf{N}})^*$ and $\mathrm{R}, \mathrm{S}, \ldots$ to range over finite sets of actions. Let $\overline{\mathrm{R}} \overset{\text{def}}{=} \{\overline{\alpha} \mid \alpha \in \mathrm{R}\}$. The syntax of BPEL *abstract activities* is defined by the following grammar:

$$
\begin{array}{rll}
P, Q, P_i ::= & \mathtt{0} & (\mathtt{empty}) \\
| & a & (\mathtt{receive}) \\
| & \overline{a} & (\mathtt{invoke}) \\
| & \sum_{i \in I} \alpha_i \,;\, P_i & (\mathtt{pick}) \\
| & P \mid_A Q & (\mathtt{flow\ \&\ link}) \\
| & P \,;\, Q & (\mathtt{sequence}) \\
| & \bigoplus_{i \in I} P_i & (\mathtt{if}) \\
| & P^{*} & (\mathtt{while})
\end{array}
$$

Each construct is called with the name of the corresponding BPEL construct. The activity $\mathtt{0}$ represents the completed process, it performs no visible action. The activity a represents the act of waiting for an incoming message. Here we take the point of view that a stands for a particular operation implemented by the process. The activity $\overline{a}$ represents the act of invoking the operation a provided by another partner. The activity $\sum_{i \in I} \alpha_i \,;\, P_i$ represents the act of waiting for any of the α_i operations to be performed, i belonging to a *finite* set I. Whichever operation α_i is performed, it first disables the remaining ones and the continuation P_i is executed. If $\alpha_i = \alpha_j$ and $i \neq j$, then the choice whether executing P_i or P_j is implementation dependent. The process $P \mid_A Q$, where A is a set of names, represents the parallel composition (`flow`) of P and Q and the creation of a private set A of `link` names that will be used by P and Q to synchronize; an example will be given shortly. The n-ary version $\prod_{i \in 1..n}^{A} P_i$ of this construct may also be considered: we stick to the binary one for simplicity. The process $P \,;\, Q$ represents the sequential composition of P followed by Q. Again we only provide a binary operator, where the BPEL one is n-ary. The process $\bigoplus_{i \in I} P_i$, again with I finite, represents an internal choice performed by the process, that results into one of the I exclusive continuations P_i. Finally, P^{*} represents the repetitive execution of process P so long as an internally verified condition is satisfied.

The pick activity $\sum_{i \in 1..n} \alpha_i \,;\, P_i$ and the if activity $\bigoplus_{i \in 1..n} P_i$ will be also written $\alpha_1 \,;\, P_1 + \cdots + \alpha_n \,;\, P_n$ and $P_1 \oplus \cdots \oplus P_n$, respectively. In the following we treat (`empty`), (`receive`), and (`invoke`) as special cases of (`pick`), while at the same time keeping the formal semantics just as easy. In particular, we write $\mathtt{0}$ for $\sum_{\alpha \in \varnothing} \alpha \,;\, P_\alpha$ and α as an abbreviation for $\sum_{\alpha \in \{\alpha\}} \alpha \,;\, \mathtt{0}$ (tailing $\mathtt{0}$ are always omitted). Let also $\mathsf{actions}(P)$ be the set of actions occurring in P.

Table 1. Legend for the operations of the BPEL process in Figure 1

Name	Operation
a	`sendPurchaseOrder`
b	`requestShipping`
c	`sendSchedule`
d	`initiatePriceCalculation`
e	`sendShippingPrice`
f	`sendInvoice`
g	`requestProductionScheduling`
h	`sendShippingSchedule`
x	`ship-to-invoce`
y	`ship-to-scheduling`

Example 1. Table 1 gives short names to the operations used in the business process shown in Figure 1. Then the whole BPEL activity can be described by the term

$$a\,;\Big(\overline{b}\,;\big((\overline{x}\,|_{\varnothing}\,c\,;\overline{y})\,|_{\{x\}}\,\overline{d}\,;x\,;\overline{e};f\big)\,|_{\{y\}}\,\overline{g}\,;y\,;\overline{h}\Big)\,;\overline{a}$$

where we use names for specifying links. Such names are restricted so that they are not visible from outside. Indeed, they are completely internal to the process and should not be visible in the process' contract. ♢

Remark 1. The BPEL specification defines a number of static analysis requirements beyond the mere syntactic correctness of processes whose purpose is to "detect any undefined semantics or invalid semantics within a process definition" [2]. Several of these requirements regard the use of links. For example, it is required that no link must cross the boundary of a repeatable construct (`while`). It is also required that link ends must be used exactly once (hence $\mathsf{0}\,|_{\{a\}}\,a$ is invalid because $\overline{a}$ is never used), and the dependency graph determined by links must be acyclic (hence $a.\overline{b}\,|_{\{a,b\}}\,b.\overline{a}$ is invalid because it contains cycles). These constraints may be implemented by restricting the arguments to the above abstract activities and then using static analysis techniques. ■

2.2 Operational Semantics of BPEL Abstract Activities

The operational semantics of BPEL abstract activities is defined by means of a completion predicate and of a labelled transition system. Let $P\checkmark$, read *P has completed*, be the least predicate such that

$$\mathsf{0}\checkmark \qquad \frac{P\checkmark \quad Q\checkmark}{P\,|_A\,Q\checkmark} \qquad \frac{P\checkmark \quad Q\checkmark}{P\,;Q\checkmark}$$

Let μ range over actions and the special name ε denote internal moves. The operational semantics of processes is described by the following rules plus the symmetric of the rules for $|$.

$$\text{(ACTION)}\quad \textstyle\sum_{i\in I}\alpha_i\,;P_i \xrightarrow{\alpha_i} P_i \qquad\qquad \text{(IF)}\quad \bigoplus_{i\in I} P_i \xrightarrow{\varepsilon} P_i$$

$$\text{(FLOW)}\quad \frac{P \xrightarrow{\mu} P' \quad \mu \notin \mathrm{A}\cup\overline{\mathrm{A}}}{P \mid_A Q \xrightarrow{\mu} P' \mid_A Q} \qquad \text{(LINK)}\quad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\overline{\alpha}} Q' \quad \alpha \in \mathrm{A}\cup\overline{\mathrm{A}}}{P \mid_A Q \xrightarrow{\varepsilon} P' \mid_A Q'}$$

$$\text{(SEQ)}\quad \frac{P \xrightarrow{\mu} P'}{P\,;Q \xrightarrow{\mu} P'\,;Q} \qquad \text{(SEQ-END)}\quad \frac{P\checkmark \quad Q \xrightarrow{\mu} Q'}{P\,;Q \xrightarrow{\mu} Q'} \qquad \text{(WHILE-END)}\quad P^* \xrightarrow{\varepsilon} \mathsf{0} \qquad \text{(WHILE)}\quad \frac{P \xrightarrow{\mu} P'}{P^* \xrightarrow{\mu} P'\,;P^*}$$

We briefly describe the rules. The process $\sum_{i\in I}\alpha_i\,;P_i$ has as many α-labelled transitions as the number of actions in $\{\alpha_i \mid i \in I\}$. After a visible transition, only the selected continuation is allowed to execute. The process $\bigoplus_{i\in I} P_i$ may internally choose to behave as one of the P_i, with $i \in I$. The process $P \mid_A Q$ allows P and Q to internally evolve autonomously, or to emit messages, or to synchronize with each other on names in the set A. It completes when both P and Q have completed. The process $P\,;Q$ reduces according to the reductions of P first, and of Q when P has completed. Finally, the process P^* may either complete in one step by reducing to $\mathsf{0}$, or it may execute P one more time followed by P^*. The choice among the two possibilities is performed internally.

Remark 2. According to the operational semantics, P^* may execute the activity P an arbitrary number of times. This is at odds with concrete BPEL activities having P^* as abstract counterpart. For example, the BPEL activity

```
<while>
  <condition> bool-expr </condition>
    activity
</while>
```

executes `activity` as long as the `bool-expr` condition is true. Representing such BPEL activity with `activity`* means *overapproximating* it. This abstraction is crucial for the decidability of our theory. ■

We illustrate the semantics of BPEL abstract activities through a couple of examples:

1. $(\overline{a} \oplus \overline{b} \mid_{\{a,b\}} a \oplus b)\,;\overline{c} \xrightarrow{\varepsilon} (\overline{a} \mid_{\{a,b\}} a \oplus b)\,;\overline{c}$ by (IF), (FLOW), and (SEQ). By the same rules, it is possible to have $(\overline{a} \mid_{\{a,b\}} a \oplus b)\,;\overline{c} \xrightarrow{\varepsilon} (\overline{a} \mid_{\{a,b\}} b)\,;\overline{c}$, which cannot reduce anymore ($\overline{a} \mid_{\{a,b\}} b$ is a *deadlocked* activity).
2. let $\Psi \overset{\text{def}}{=} \mathsf{0}\,;(\mathsf{0}\oplus\mathsf{0})^*$. Then, according to rules (SEQ-END), (IF), and (WHILE), $\Psi \xrightarrow{\varepsilon} \Psi$ and $\Psi \xrightarrow{\varepsilon} \mathsf{0}$.

Let $\overset{\varepsilon}{\Longrightarrow}$ be the reflexive, transitive closure of $\xrightarrow{\varepsilon}$ and $\overset{\alpha}{\Longrightarrow}$ be $\overset{\varepsilon}{\Longrightarrow}\xrightarrow{\alpha}\overset{\varepsilon}{\Longrightarrow}$; let also $P \xrightarrow{\mu}$ (resp. $P \overset{\alpha}{\Longrightarrow}$) if there exists P' such that $P \xrightarrow{\mu} P'$ (resp. $P \overset{\alpha}{\Longrightarrow} P'$); we let $P \not\xrightarrow{\mu}$ if not $P \xrightarrow{\mu}$.

A relevant property of our BPEL abstract calculus is that the model of every activity is always finite. This result is folklore (the argument is similar to the one for CCS$_*$ [7]).

Lemma 1. *Let Reach*$(P) = \{Q \mid$ *there are* $\mu_1, \dots, \mu_n$ *with* $P \xrightarrow{\mu_1} \cdots \xrightarrow{\mu_n} Q\}$. *Then, for every activity* P*, the set Reach*(P) *is always finite.*

We introduce a number of auxiliary definitions that will be useful in Section 3. By Lemma 1 these notions are trivially decidable.

Definition 1. *We introduce the following notation:*

- $P\!\uparrow$ *if there is an infinite sequence of* ε*-transitions* $P \xrightarrow{\varepsilon}\xrightarrow{\varepsilon} \cdots$ *starting from* P*. Let* $P\!\downarrow$ *if not* $P\!\uparrow$*.*
- $\mathsf{init}(P) \stackrel{\text{def}}{=} \{\alpha \mid P \stackrel{\alpha}{\Longrightarrow}\}$;
- *we say that* P *has* ready set R*, notation* $P \Downarrow \mathrm{R}$*, if* $P \stackrel{\varepsilon}{\Longrightarrow} P'$ *and* $\mathrm{R} = \mathsf{init}(P')$;
- *let* $P \stackrel{\alpha}{\Longrightarrow}$*. Then* $P(\alpha) \stackrel{\text{def}}{=} \bigoplus_{P \stackrel{\varepsilon}{\Longrightarrow}\xrightarrow{\alpha} P'} P'$*. We call* $P(\alpha)$ *the* continuation of P after α*.*

The above definitions are almost standard, except for $P(\alpha)$ (that we already used in [17]). Intuitively, $P(\alpha)$ represents the residual behavior of P after an action α, from the point of view of the party that is interacting with P. Indeed, the party does not know which, of the possibly multiple, α-labelled branches P has taken. For example $(a\,;b + a\,;c + b\,;d)(a) = b \oplus c$ and $(a\,;b + a\,;c + b\,;d)(b) = d$.

2.3 The Compliance Preorder

We proceed defining a notion of equivalence between activities that is based on their observable behavior. To this aim, we introduce a special name e for denoting the successful termination of an activity ("e" stands for `end`). By *compliance* between a "client" activity T and a "service" activity P we mean that every interaction between T and P where P stops communicating with T is such that T has reached a successfully terminated state.

Definition 2 (Compliance). *Let* $A_P = \{a \mid a \in \mathsf{actions}(P) \cup \overline{\mathsf{actions}(P)}\}$ *and* $\mathsf{e} \notin A_P$*. The (client) activity* T *is* compliant with *the (service) activity* P*, written* $T \dashv P$*, if* $P \mid_{A_P} T \stackrel{\varepsilon}{\Longrightarrow} P' \mid_{A_P} T'$ *implies:*

1. *if* $P' \mid_{A_P} T' \stackrel{\varepsilon}{\not\longrightarrow}$*, then* $\{\mathsf{e}\} \subseteq \mathsf{init}(T')$*, and*
2. *if* $P'\!\uparrow$*, then* $\{\mathsf{e}\} = \mathsf{init}(T')$*.*

We write $P \sqsubseteq Q$*, called* compliance preorder*, if and only if* $T \dashv P$ *implies* $T \dashv Q$ *for every* T*. Let* $\eqsim \stackrel{\text{def}}{=} \sqsubseteq \cap \sqsupseteq$*.*

According to the notion of compliance, if the client-service conversation terminates, then the client is in a successful state (it will emit an e-name). For example, $a\,;\mathsf{e} + b\,;\mathsf{e} \dashv \overline{a} \oplus \overline{b}$ and $a\,;\mathsf{e} \oplus b\,;\mathsf{e} \dashv \overline{a} + \overline{b}$ but $a\,;\mathsf{e} \oplus b\,;\mathsf{e} \not\dashv \overline{a} \oplus \overline{b}$ because of the computation $a\,;\mathsf{e} \oplus b\,;\mathsf{e} \mid_{\{a,b\}} \overline{a} \oplus \overline{b} \Longrightarrow a\,;\mathsf{e} \mid_{\{a,b\}} \overline{b} \not\longrightarrow$ where the client waits for an interaction on a in vain. Similarly, the client must reach a successful state if the conversation does not terminate but the divergence is due to the service. In this case, however, every reachable state of the client must be such that the only possible action is e. The practical justification of such a notion of compliance derives from the fact that connection-oriented communication protocols (like those used for interaction with Web services) typically provide for an explicit end-of-connection signal. Consider for example the client behavior $\mathsf{e} + \overline{a} \,;\, \mathsf{e}$. Intuitively this client tries to send a request on the name a, but it can also succeed if the service rejects the request. So $\mathsf{e} + \overline{a} \,;\, \mathsf{e} \dashv \mathsf{0}$ because the client can detect the fact that the service is not ready to interact on a. The same client interacting with a diverging service would have no way to distinguish a service that is taking a long time to accept the request from a service that is perpetually performing internal computations, hence $\mathsf{e} + \overline{a} \,;\, \mathsf{e} \not\dashv \Psi$. As a matter of facts, the above notion of compliance makes Ψ the "smallest service" – the one a client can make the least number of assumptions on (this property will be fundamental in the definition of principal dual contract in Section 4). That is $\Psi \sqsubseteq P$, for every P. As another example, we notice that $a \,;\, b + a \,;\, c \sqsubseteq a \,;\, (b \oplus c)$ since, after interacting on a, a client of the smaller service is not aware of which state the service is in (it can be either b or c). Had we picked only one a-derivative of the smaller contract behavior, we would have failed to relate it with the a-derivative of the larger contract, since both $b \not\sqsubseteq b \oplus c$ and $c \not\sqsubseteq b \oplus c$.

As by Definition 2, it is difficult to understand the general properties of the compliance preorder because of the universal quantification over all (client) activities T. For this reason, it is convenient to provide an alternative characterization of $\sqsubseteq$ which turns out to be the following:

Definition 3. *A* coinductive compliance *is a relation $\mathcal{R}$ such that $P \;\mathcal{R}\; Q$ and $P\!\downarrow$ implies*

1. *$Q\!\downarrow$, and*
2. *$Q \Downarrow \mathrm{R}$ implies $P \Downarrow \mathrm{S}$ for some $\mathrm{S} \subseteq \mathrm{R}$, and*
3. *$Q \stackrel{\alpha}{\Longrightarrow}$ implies $P \stackrel{\alpha}{\Longrightarrow}$ and $P(\alpha) \;\mathcal{R}\; Q(\alpha)$.*

Let $\preceq$ be the largest coinductive compliance relation.

The pre-order $\preceq$ corresponds to the *must-testing preorder* [15] and is also an alternative definition of $\sqsubseteq$:

Theorem 1. *$P \sqsubseteq Q$ if and only if $P \preceq Q$.*

3 Contracts

In this section we discuss how to associate a behavioral description, called *contract*, to a BPEL abstract activity. The ultimate goal is being able to reason about properties of BPEL activities by means of the respective contracts.

Contracts use a set of contract names, ranged over $\mathsf{C}, \mathsf{C}', \mathsf{C}_1, \ldots$. A contract is a tuple

$$(\mathsf{C}_1 = \sigma_1, \ldots, \mathsf{C}_n = \sigma_n,\ \sigma)$$

where $\mathsf{C}_j = \sigma_j$ are contract name definitions, σ is the main term, and we assume that there is no chain of definitions of the form $\mathsf{C}_{n_1} = \mathsf{C}_{n_2}$, $\mathsf{C}_{n_2} = \mathsf{C}_{n_3}$, $\dots$, $\mathsf{C}_{n_k} = \mathsf{C}_{n_1}$. The syntax of σ_j and σ is given by

$$\sigma \quad ::= \quad \mathsf{C} \quad | \quad \alpha\,;\sigma \quad | \quad \sigma + \sigma \quad | \quad \sigma \oplus \sigma$$

where $\mathsf{C} \in \{\mathsf{C}_1, \dots, \mathsf{C}_n\}$. The contract $\alpha\,;\sigma$ represents sequential composition in the restricted form of prefixing. The operations $+$ and $\oplus$ correspond to `pick` and `if` of BPEL activities, respectively. These operations are assumed to be associative and commutative; therefore we will write $\sigma_1 + \cdots + \sigma_n$ and $\sigma_1 \oplus \cdots \oplus \sigma_n$ without confusion and will sometimes shorten these contracts as $\sum_{i\in 1..n} \sigma_i$ and $\bigoplus_{i\in 1..n} \sigma_i$, respectively. The contract name C is used to model recursive behaviors such as $\mathsf{C} = a\,;\mathsf{C}$. In what follows we will leave contract name definitions implicit and identify a contract $(\mathsf{C}_1 = \sigma_1, \dots, \mathsf{C}_n = \sigma_n,\ \sigma)$ with its main body σ. We will write $\mathsf{cnames}(\sigma)$ for the set $\{\mathsf{C}_1, \dots, \mathsf{C}_n\}$ and use the following abbreviations:

- $\mathsf{0} \overset{\text{def}}{=} \mathsf{C}_\mathsf{0}$, where $\mathsf{C}_\mathsf{0} = \mathsf{C}_\mathsf{0} + \mathsf{C}_\mathsf{0}$ represents a terminated activity;
- $\Omega \overset{\text{def}}{=} \mathsf{C}_\Omega$, where $\mathsf{C}_\Omega = \mathsf{C}_\Omega \oplus \mathsf{C}_\Omega$ represents divergence, that is a non-terminating activity.

Note that, even if apparently simpler, the contract language *is not* a sublanguage of BPEL abstract activities. For example, Ω cannot be written as a term in the syntax of Section 2.1. Nevertheless, in the following, we demonstrate that contracts provide alternative descriptions (with respect to the preorder $\precsim$) to BPEL abstract activities.

The operational semantics of contracts is defined by the rules below:

$$\alpha\,;\sigma \xrightarrow{\alpha} \sigma \qquad \sigma \oplus \rho \xrightarrow{\varepsilon} \sigma$$

$$\frac{\sigma \xrightarrow{\varepsilon} \sigma'}{\sigma + \rho \xrightarrow{\varepsilon} \sigma' + \rho} \qquad \frac{\sigma \xrightarrow{\alpha} \sigma'}{\sigma + \rho \xrightarrow{\alpha} \sigma'} \qquad \frac{\mathsf{C} = \sigma \quad \sigma \xrightarrow{\mu} \sigma'}{\mathsf{C} \xrightarrow{\mu} \sigma'}$$

plus the symmetric of rules $+$ and $\oplus$. Note that $+$ evaluates the branches as long as they can perform invisible actions. This rule is absent in BPEL abstract activities because, there, the branches are always guarded by an action.

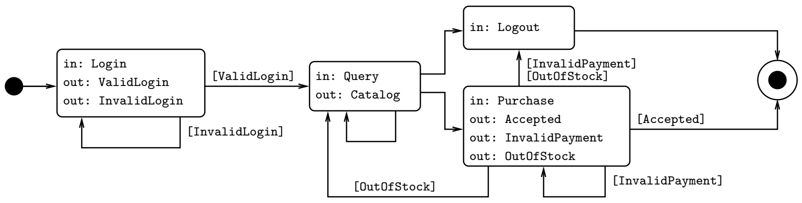


Fig. 2. Contract of a simple e-commerce service as a WSCL diagram

Example 2. The Web service conversation language WSCL [3] describes *conversations* between two parties by means of an activity diagram (Figure 2). The diagram is made of *interactions* connected with each other by *transitions*. An interaction is a basic one-way or two-way communication between the client and the server. Two-way communications are just a shorthand for two sequential one-way interactions. Each interaction has a *name* and a list of *document types* that can be exchanged during its execution. A transition connects a *source* interaction with a *destination* interaction. A transition may be *labeled* by a document type if it is active only when a message of that specific document type was exchanged during the previous interaction.

The diagram in Figure 2 describes the conversation of a service requiring clients to login before they can issue a query. After the query, the service returns a catalog. From this point on, the client can decide whether to purchase an item from the catalog or to logout and leave. In case of purchase, the service may either report that the purchase is successful, or that the item is out-of-stock, or that the client's payment is refused. By interpreting names as message types, this e-commerce service can be described by the tuple:

$$
\begin{aligned}
(\ \ & \mathsf{C}_1 = \texttt{Login}\,;(\overline{\texttt{InvalidLogin}}\,;\mathsf{C}_1 \oplus \overline{\texttt{ValidLogin}}\,;\mathsf{C}_2)\ , \\
& \mathsf{C}_2 = \texttt{Query}\,;\overline{\texttt{Catalog}}\,;(\mathsf{C}_2 + \mathsf{C}_3 + \mathsf{C}_4)\ , \\
& \mathsf{C}_3 = \texttt{Purchase}\,;(\,\overline{\texttt{Accepted}} \\
& \qquad\qquad\qquad \oplus \overline{\texttt{InvalidPayment}}\,;(\mathsf{C}_3 + \mathsf{C}_4) \\
& \qquad\qquad\qquad \oplus \overline{\texttt{OutOfStock}}\,;(\mathsf{C}_2 + \mathsf{C}_4)\,)\ , \\
& \mathsf{C}_4 = \texttt{Logout}\ , \\
& \mathsf{C}_1\ \ )
\end{aligned}
$$

There is a strict correspondence between unlabeled (respectively, labeled) transitions in Figure 2 and external (respectively, internal) choices in the contract. Recursion is used for expressing iteration (the cycles in the figure) so that the client is given another chance whenever an action fails for some reason. ◊

We can relate BPEL abstract activities and contracts by means of the corresponding transition systems. To this aim, let X and Y range over BPEL abstract activities *and* contracts. Then, X and Y interact according to the rules

$$
\frac{\mathsf{X} \xrightarrow{\varepsilon} \mathsf{X}'}{\mathsf{X} \,\|\, \mathsf{Y} \xrightarrow{\varepsilon} \mathsf{X}' \,\|\, \mathsf{Y}}
\qquad
\frac{\mathsf{Y} \xrightarrow{\varepsilon} \mathsf{Y}'}{\mathsf{X} \,\|\, \mathsf{Y} \xrightarrow{\varepsilon} \mathsf{X} \,\|\, \mathsf{Y}'}
\qquad
\frac{\mathsf{X} \xrightarrow{\alpha} \mathsf{X}' \quad \mathsf{Y} \xrightarrow{\overline{\alpha}} \mathsf{Y}'}{\mathsf{X} \,\|\, \mathsf{Y} \xrightarrow{\varepsilon} \mathsf{X}' \,\|\, \mathsf{Y}'}
$$

It is possible to extend the definition of compliance to contracts and, by Definition 2, obtain a relation that allows us to compare activities and contracts without distinction, and similarly for $\preceq$. To be precise, the relation $\mathsf{X} \sqsubseteq \mathsf{Y}$ is smaller (in principle) than the relation $\sqsubseteq$ given in Definition 2 because, as we have said, the contract language is not a sublanguage of that of activities and, therefore, the set of tests that can be used for comparing X and Y is larger. Nonetheless, the process language used in [12] includes both BPEL abstract activities and contracts and since $\sqsubseteq$ is equivalent to must-testing, then we may safely use the same symbol $\sqsubseteq$ for both languages. This is a key point in our argument, which will allow us to define, for every activity P, a contract σ_P such that $P \eqsim \sigma_P$. In particular, let C_P be the set of contract name definitions defined as follows

$$\mathsf{C}_P = \begin{cases} \Omega & \text{if } P\uparrow \\ \bigoplus_{P \Downarrow \mathrm{R}} \sum_{\alpha \in \mathrm{R}} \alpha \,;\, \mathsf{C}_{P(\alpha)} & \text{otherwise} \end{cases}$$

A relevant property of C_P is an immediate consequence of Lemma 1.

Lemma 2. *For every P, the set* $\mathsf{cnames}(\mathsf{C}_P)$ *is finite.*

The construction of the contract C_P with respect to a BPEL abstract activity is both correct and complete with respect to compliance:

Theorem 2. $P \eqsim \mathsf{C}_P$.

4 Service Discovery and Dual Contracts

We now turn our attention to the problem of querying a database of Web service contracts. To this aim, the relation $\sqsubseteq$ (and the must-testing) turns out to be too strong (see below). Following [17], we switch to more informative service contracts than what described in Section 3. In particular, we consider pairs $\mathrm{I} : \sigma$, where I is the interface, i.e. the set of actions performed by the service, and σ is as in Section 3 (it is intended that the names occurring in σ are included into I). It is reasonable to think that a similar extension applies to client contracts: clients, which are defined by BPEL activities as well, are abstracted by terms in the language of Section 2 and, in turn, their behavior is defined by a term in the contract language, plus the interface.

Definition 4 (Subcontract relation). *Let* $\mathrm{I} : \sigma \lesssim \mathrm{J} : \tau$ *if* $\mathrm{I} \subseteq \mathrm{J}$ *and, for every* ρ *such that* $\mathsf{actions}(\rho) \setminus \{\mathsf{e}\} \subseteq \overline{\mathrm{I}}$ *and* $\rho \dashv \sigma$ *implies* $\rho \dashv \tau$. *Let* $\approx$ *be* $\lesssim \cap \gtrsim$.

Let us comment on the differences between $\mathrm{I} : \sigma \lesssim \mathrm{J} : \tau$ and $\sigma \sqsubseteq \tau$. We notice that $\mathrm{I} : \sigma \lesssim \mathrm{J} : \tau$ only if $\mathrm{I} \subseteq \mathrm{J}$. This apparently natural prerequisite has substantial consequences on the properties of $\lesssim$ because it ultimately enables width and depth extensions, which are not possible in the $\sqsubseteq$ preorder. For instance, we have $\{a\} : a \lesssim \{a, b\} : a + b$ whilst $a \not\sqsubseteq a + b$ (width extension). Similarly we have $\{a\} : a \lesssim \{a, b\} : a \,;\, b$ whilst $a \not\sqsubseteq a \,;\, b$ (depth extension). These extensions are desirable when searching for services, since every service offering more methods than required is a reasonable result of a query. The precise relationship between $\lesssim$ and $\sqsubseteq$ is expressed by the following statement.

Proposition 1. $\mathrm{I} : \sigma \approx \mathrm{J} : \tau$ *if and only if* $\sigma \eqsim \tau$ *and* $\mathrm{I} = \mathrm{J}$.

The basic problem for querying Web service repositories is that, given a client $\mathrm{K} : \rho$, one wishes to find all the service contracts $\mathrm{I} : \sigma$ such that $\mathsf{actions}(\rho) \setminus \{\mathsf{e}\} \subseteq \overline{\mathrm{I}}$ and $\rho \dashv \sigma$. We attack this problem in two steps: first of all, we compute *one* *particular service contract* $\overline{\mathrm{K}} \setminus \{\overline{\mathsf{e}}\} : \mathsf{D}^{\mathrm{K}}_{\rho}$ such that $\rho \dashv \mathsf{D}^{\mathrm{K}}_{\rho}$; second, we take all the services in the registry whose contract is larger than this one. In order to maximize the number of service contracts returned as answer to the query, the *dual* of a (client) contract $\mathrm{K} : \rho$ should be a contract $\overline{\mathrm{K}} \setminus \{\overline{\mathsf{e}}\} : \mathsf{D}^{\mathrm{K}}_{\rho}$ such that it is the smallest service contract that satisfies the client contract $\mathrm{K} : \rho$. We call such contract the *principal dual contract* of $\mathrm{K} : \rho$.

In defining the principal dual contract, it is convenient to restrict the definition to those client's behaviors ρ that never lead to $\mathsf{0}$ without emitting e. For example, the behavior $a\,;\mathsf{e} + b$ describes a client that succeeds if the service proposes $\overline{a}$, but that fails if the service proposes $\overline{b}$. As far as querying is concerned, such behavior is completely equivalent to $a\,;\mathsf{e}$. As another example, the degenerate client behavior $\mathsf{0}$ is such that no service will ever satisfy it. In general, if a client is unable to handle a particular action, like b in the first example, it should simply omit that action from its behavior. We say that a (client) contract $\mathrm{K} : \rho$ is *canonical* if, whenever $\rho \overset{\varphi}{\Longrightarrow} \rho'$ is maximal, then $\varphi = \varphi'\mathsf{e}$ and $\mathsf{e} \notin \mathsf{actions}(\varphi')$. For example $\{a, \mathsf{e}\} : a\,;\mathsf{e}$, $\{a\} : \mathsf{C}$, where $\mathsf{C} = a\,;\mathsf{C}$, and $\varnothing : \Omega$ are canonical; $\{a, b, \mathsf{e}\} : a\,;\mathsf{e} + b$ and $\{a\} : \mathsf{C}'$, where $\mathsf{C}' = a \oplus \mathsf{C}'$, are not canonical.

Observe that Lemma 1 also applies to contracts. Therefore it is possible to extend the notions in Definition 1, by replacing activities with contracts.

Definition 5 (Dual contract). *Let* $\mathrm{K} : \rho$ *be a canonical contract. The* dual *of* $\mathrm{K} : \rho$ *is* $\overline{\mathrm{K}} \setminus \{\overline{\mathsf{e}}\} : \mathsf{D}^{\mathrm{K}}_{\rho}$ *where* $\mathsf{D}^{\mathrm{K}}_{\rho}$ *is the contract name defined as follows:*

$$\mathsf{D}^{\mathrm{K}}_{\rho} \overset{\text{def}}{=} \begin{cases} \Omega & \textit{if } \mathsf{init}(\rho) = \{\mathsf{e}\} \\[2ex] \sum_{\substack{\rho \Downarrow \mathrm{R} \\ \mathrm{R}\setminus\{\mathsf{e}\} \neq \varnothing}} \Big(\underbrace{\mathsf{0} \oplus}_{\textit{if } \mathsf{e} \in \mathrm{R}} \bigoplus_{\alpha \in \mathrm{R}\setminus\{\mathsf{e}\}} \overline{\alpha}\,; \mathsf{D}^{\mathrm{K}}_{\rho(\alpha)} \Big) + \mathtt{OTH}_{\mathrm{K}\setminus\mathsf{init}(\rho)} & \textit{otherwise} \end{cases}$$

$$\text{where } \mathtt{OTH}_{\mathrm{S}} \overset{\text{def}}{=} \mathsf{0} \underbrace{\oplus \bigoplus_{\alpha \in \mathrm{S}} \overline{\alpha}\,; \Omega}_{\textit{if } \mathrm{S} \neq \varnothing}$$

Few comments about $\mathsf{D}^{\mathrm{K}}_{\rho}$, when $\mathsf{init}(\rho) \neq \{\mathsf{e}\}$, follow. In this case, the behavior ρ may autonomously transit to different states, each one offering a particular ready set. Thus the dual behavior leaves the choice to the client: this is the reason for the external choice in the second line. Once the state has been chosen, the client offers to the service a spectrum of possible actions: this is the reason for the internal choice underneath the sum $\sum$.

The contract $\mathtt{OTH}_{\mathrm{K}\setminus\mathsf{init}(\rho)}$ covers all the cases of actions that are allowed by the interface and that are not offered by the client. The point is that the dual operator must compute the principal (read, the smallest) service contract that satisfies the client, and the smallest convergent behavior with respect to a nonempty (finite) interface S is $\mathsf{0} \oplus \bigoplus_{\alpha \in \mathrm{S}} \overline{\alpha}\,; \Omega$. The $\mathsf{0}$ summand accounts for the possibility that none of the actions in $\mathrm{K} \setminus \mathsf{init}(\rho)$ is present. The external choice "+" distributes the proper dual contract over the internal choice of all the actions in $\mathrm{K} \setminus \mathsf{init}(\rho)$. For example, $\mathsf{D}^{\{a,\overline{a},\mathsf{e}\}}_{a\,;\,\mathsf{e}} = \overline{a}\,; \Omega + (\mathsf{0} \oplus a\,; \Omega)$. The dual of a divergent (canonical) client $\{\text{A}\} : \mathsf{C}$, where $\mathsf{C} = a \,;\, \mathsf{e} \oplus \mathsf{C}$, is also well defined: $\mathsf{D}^{\{a,\mathsf{e}\}}_{\mathsf{C}''} = \overline{a} \,;\, \Omega$. We finally observe that the definition also accounts for duals of nonterminating clients, such as $\{\text{A}\} : \mathsf{C}'$, where $\mathsf{C}' = a \,;\, \mathsf{C}'$. In this case, $\mathsf{D}^{\{a\}}_{\mathsf{C}'} = \overline{a} \,;\, \mathsf{D}^{\{a\}}_{\mathsf{C}'}$.

Similarly to the definition of contract names C_P, it is possible to prove that $\mathsf{D}^{\text{K}}_{\rho}$ is well defined.

Lemma 3. *For every* $\text{K} : \rho$*, the set* $\mathsf{cnames}(\mathsf{D}^{\text{K}}_{\rho})$ *is finite.*

The property that $\overline{\text{K}} \setminus \{\overline{\mathsf{e}}\} : \mathsf{D}^{\text{K}}_{\rho}$ is the least dual contract of $\text{K} : \rho$ follows.

Theorem 3. *Let* $\text{K} : \rho$ *be a canonical contract. Then:*

1. $\rho \dashv \mathsf{D}^{\text{K}}_{\rho}$;
2. *if* $\overline{\text{K}} \setminus \{\overline{\mathsf{e}}\} \subseteq \text{S}$ *and* $\rho \dashv \sigma$*, then* $\overline{\text{K}} \setminus \{\overline{\mathsf{e}}\} : \mathsf{D}^{\text{K}}_{\rho} \lesssim \text{S} : \sigma$.

A final remark is about the computational complexity of the discovery algorithm. Determining $\lesssim$ is EXPTIME-complete in the size of the contracts [1], which has to be multiplied by the number of $\lesssim$-checks (to find a compliant service in the repository) to obtain the overall cost.

5 Conclusions

In this contribution we have studied a formal theory of Web service abstract (behavioral) definitions as normal forms of a natural semantics for BPEL activities. Our abstract definitions may be effectively used in any query-based system for service discovery because they support a notion of principal dual contract. This operation is currently done in an ad hoc fashion using search engines or similar technologies.

Several future research directions stem from this work. On the technical side, a limit of our technique is that BPEL activities are "static", *i.e.* they cannot create other services on the fly. This constraint implies the finiteness of models and, for this reason, it is possible to effectively associate an abstract description to activities. However, this impacts on scalability, in particular when services adapt to peaks of requests by creating additional services. It is well-known that such an additional feature makes models to be infinite states and requires an approximate inferential process to extract abstract descriptions from activities. Said otherwise, extending our technique to full CCS or π-calculus amounts to defining abstract finite models such that Theorem 2 does not hold anymore. For this reason, under- and over-estimations for services and clients, respectively, must be provided.

Another interesting technical issue concerns the extension of our study to other semantics for BPEL activities, such as the preorder in [6], or even to weak bisimulation (which has a polynomial computational cost). Perhaps one may use axiomatizations of these equivalences for determining the class of contracts. However it is not clear whether they admit a principal dual contract or not.

It is also interesting to prototyping our theory and experimenting it on some existing repository, such as `http://www.service-repository.com/`. To this aim we might use tools that have been already developed for the must testing, such as the concurrency workbench [11].

References

1. Aceto, L., Ingolfsdottir, A., Srba, J.: The algoritmics of bisimilarity. In: Sangiorgi, D., Rutten, J. (eds.) Advanced Topics in Bisimulation and Coinduction. Cambridge Tracts in Theoretical Computer Science, vol. 52, ch.3, pp. 100–172. Cambridge University Press (2011)
2. Alves, A., et al.: Web Services Business Process Execution Language Version 2.0 (January 2007), `http://docs.oasis-open.org/wsbpel/2.0/CS01/wsbpel-v2.0-CS01.html`
3. Banerji, A., Bartolini, C., Beringer, D., Chopella, V., et al.: Web Services Conversation Language (WSCL) 1.0 (March 2002), `http://www.w3.org/TR/2002/NOTE-wscl10-20020314`
4. Benzaken, V., Castagna, G., Frisch, A.: CDuce: an XML-centric general-purpose language. SIGPLAN Notices 38(9), 51–63 (2003)
5. Beringer, D., Kuno, H., Lemon, M.: Using WSCL in a UDDI Registry 1.0. UDDI Working Draft Best Practices Document (2001), `http://xml.coverpages.org/HP-UDDI-wscl-5-16-01.pdf`
6. Bravetti, M., Zavattaro, G.: A theory of contracts for strong service compliance. Mathematical Structures in Computer Science 19(3), 601–638 (2009)
7. Busi, N., Gabbrielli, M., Zavattaro, G.: On the expressive power of recursion, replication and iteration in process calculi. Mathematical Structures in Computer Science 19(6), 1191–1222 (2009)
8. Castagna, G., Gesbert, N., Padovani, L.: A Theory of Contracts for Web Services. ACM Transactions on Programming Languages and Systems 31(5) (2009)
9. Chaki, S., Rajamani, S.K., Rehof, J.: Types as models: model checking message-passing programs. SIGPLAN Not. 37(1), 45–57 (2002)
10. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Services Description Language (WSDL) 1.1 (2001), `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`
11. Cleaveland, R., Parrow, J., Steffen, B.: The concurrency workbench: a semantics-based tool for the verification of concurrent systems. ACM Trans. Program. Lang. Syst. 15(1), 36–72 (1993)
12. De Nicola, R., Hennessy, M.: Testing equivalences for processes. Theor. Comput. Sci. 34, 83–133 (1984)
13. De Nicola, R., Hennessy, M.: CCS without τ's. In: Ehrig, H., Levi, G., Montanari, U. (eds.) CAAP 1987 and TAPSOFT 1987. LNCS, vol. 249, pp. 138–152. Springer, Heidelberg (1987)
14. Gay, S., Hole, M.: Subtyping for session types in the π-calculus. Acta Informatica 42(2-3), 191–225 (2005)
15. Hennessy, M.: Algebraic Theory of Processes. Foundation of Computing. MIT Press (1988)
16. Hosoya, H., Pierce, B.C.: XDuce: A statically typed XML processing language. ACM Trans. Internet Techn. 3(2), 117–148 (2003)
17. Laneve, C., Padovani, L.: The must preorder revisited – an algebraic theory for web services contracts. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 212–225. Springer, Heidelberg (2007)
18. Milner, R.: A Calculus of Communicating Systems. Springer (1982)
19. Nielson, H.R., Nielson, F.: Higher-order concurrent programs with finite communication topology (extended abstract). In: Proceedings of POPL 1994, pp. 84–97. ACM Press (1994)
20. Parastatidis, S., Webber, J.: MEP SSDL Protocol Framework (April 2005), `http://ssdl.org`

A Compositional Automata-Based Semantics for Property Patterns

Kalou Cabrera Castillos[1], Frédéric Dadeau[1], Jacques Julliand[1], Bilal Kanso[2], and Safouan Taha[2]

[1] FEMTO-ST/DISC - INRIA CASSIS Project
16 route de Gray 25030 Besançon cedex, France
[2] SUPELEC Systems Sciences (E3S) - Computer Science Department
3 rue Joliot-Curie F-91192 Gif-sur-Yvette cedex, France
{kalou.cabrera,frederic.dadeau,jacques.julliand}@femto-st.fr,
{bilal.kanso,safouan.taha}@supelec.fr

Abstract. Dwyer et al. define a language to specify dynamic properties based on predefined patterns and scopes. To define a property, the user has to choose a pattern and a scope among a limited number of them. Dwyer et al. define the semantics of these properties by translating each composition of a pattern and a scope into usual temporal logics (LTL, CTL, etc.). First, this translational semantics is not compositional and thus not easily extensible to other patterns/scopes. Second, it is not always faithful to the natural semantics of the informal definitions.

In this paper, we propose a compositional automata-based approach defining the semantics of each pattern and each scope by an automaton. Then, we propose a composition operation in such a way that the property semantics is defined by composing the automata. Hence, the semantics is compositional and easily extensible as we show it by handling many extensions to the Dwyer et al.'s language. We compare our compositional semantics with the Dwyer et al.'s translational semantics by checking whether our automata are equivalent to the Büchi automata of the LTL expressions given by Dwyer et al. In some cases, our semantics reveals a lack of homogeneity within Dwyer et al.'s semantics.

Keywords: Formal Methods, Temporal Properties, Compositional Automata Semantics, Temporal logics, Property Patterns.

1 Motivations

Dynamic properties are commonly described by temporal logics such as the Linear Temporal Logic (LTL). These formalisms are difficult to appropriate by system designers and validation engineers. In order to ease their understanding and writing, Dwyer et al. (denoted DAC in the reminder of the paper) propose in [4,5] a language of properties based on the composition of predefined patterns and scopes. A pattern expresses a temporal property on executions seen as sequences of states/events. A scope determines the parts of executions on which the pattern must hold.

In these works, we can measure how much it can be difficult to express properties directly by temporal formulæ. For example, with the language of DAC, one can express the following property: "the state property P' responds to the state property P between the state properties Q and R" by composing the pattern P' responds to P and the scope between Q and R. The corresponding LTL formula given by DAC is: $\Box((Q \wedge \neg R \wedge \Diamond R) \Rightarrow (P \Rightarrow (\neg R \ \mathsf{U}\ (P' \wedge \neg R)))\ \mathsf{U}\ R)$. Even if the specifier is familiar with LTL, these formulæ are very difficult either to write or to understand due to the huge semantics gap between the intuitive formulation of the property in the natural language and its complex and error-prone translation into LTL.

Besides the natural semantics of the patterns and scopes, DAC provide formal semantics by translation into many temporal logics, mapping each pattern/scope combination to a corresponding temporal formula. As there are 10 patterns and 5 scopes, they had to translate the 50 combinations [3]. DAC also noted many possible patterns/scopes variants [3], but they do not support them because translating more than 20 pattern and 20 scope variants requires up to 400 temporal formulæ. Moreover, DAC have defined informally generic patterns (e.g. a first chain of events precedes a second chain of events) that they do not succeed to translate into equivalent generic temporal logic formulæ. Hence, they translated a limited number of obvious cases (e.g. chains having only 1 or 2 events). **Extensibility** and **Genericity** are the main limitations of such a translational semantics.

Furthermore, this translational semantics arises two consistency limitations:

Faithfulness : DAC claim that the temporal formulæ were primarily validated by peer review amongst the project members and then tested against some (un)satisfying sequences of states/events. Hence, we have no formal guarantee that the translated temporal formula is faithful to the intended natural semantics associated to the pattern/scope combination;
Homogeneity : DAC define their language by clearly separating both pattern and scope notions. From a user point of view, a pattern (resp. scope) has a unique natural semantics never mind the scope (resp. pattern) with which it is combined. By adopting translational semantics, they flattened this key separation and translated each pattern and each scope many times into different formulæ corresponding to the different possible combinations. Hence, the same pattern (resp. scope) may have different interpretations according to the scope (resp. pattern) with which it is combined.

In this work, we want a specification language (1) to make easier the expression of the temporal properties by relying on the predefined patterns and scopes of DAC [5]. This language must be easily extensible (2) by adding new variants of patterns and scopes thanks to a compositional semantics. Finally, we intend to adopt an automata-based semantics (3) that is well-adapted to verify properties, and to generate and evaluate tests because it is provided with many usual structural coverage criteria.

These motivations bring three main contributions that we present in this paper. First, we define a compositional semantics giving an automaton semantics combining the automata of any pattern and any scope. Second, we compare this compositional semantics w.r.t. the LTL translational semantics given by DAC. We will show that even though they focused on translating few specification patterns, they give non-homogeneous interpretations to some patterns and scopes when writing the LTL formulæ. Third, We will give support to many generic patterns and many scope variants emphasizing the extensibility of our compositional semantics.

The paper is structured as follows. Sec. 2 recalls the property language proposed by DAC Sec. 3 presents the compositional semantics of this language by means of automata and their composition. Sec. 4 compares our semantics w.r.t. DAC's semantics and presents the automatization process of our approach using an LTL transformation tool into Büchi automata and a model-checking environment to prove that our automaton is (or is not) equivalent to the LTL formula. Sec. 5 shows the extensibility potential of the language and its semantics. Finally, Sec. 6 concludes and gives some future works.

2 Dwyer et al.'s Property Specification Language

DAC have proposed a pattern-based approach [4]. This approach uses specification patterns that, at a higher abstraction level, capture recurring temporal properties. The main idea is that a temporal property is a combination of one **pattern** and one **scope**. A scope is the part of system execution paths over which a pattern must hold.

Patterns. The patterns are temporal conditions on the system executions. DAC propose the ten following patterns classified in the left side of Fig. 1.

- always P: the state property P must hold in all states,
- never P: the state property P does not occur in any state,
- eventually P: the state property P occurs at least once,
- eventually P at most 2 times: the state property P becomes true (after being false in the preceding state) at most 2 times. In other words, switching from $\neg P$ to P occurs at most twice
- P precedes P′: a state property P′ must always be preceded by a state property P within the execution interval,
- (P_1, P_2) precedes P′: a state property P′ must be preceded by a sequence of states starting by a state property P_1 and leading to a state property P_2,
- P precedes (P'_1, P'_2): a sequence of state properties P'_1, P'_2 must be preceded by a state property P,
- P′ responds to P: a state property P must always be followed by a state property P′ within the execution interval,
- P′ responds to (P_1, P_2): a sequence of states starting by a state property P_1 and leading to a state property P_2 must be followed by a state property P′,
- (P'_1, P'_2) responds to P: a state property P must be followed by a sequence of states P'_1, P'_2.

Scopes. A scope determines the system execution intervals over which the pattern must hold. In [4], the authors propose five kinds of scopes that are illustrated

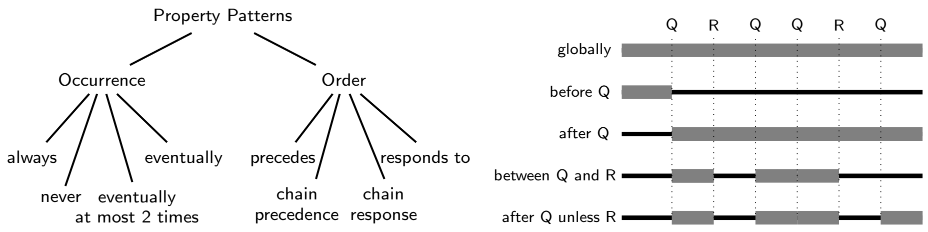


Fig. 1. DAC's Patterns and Scopes

in the right part of Fig. 1. A property is true if the pattern holds on the execution intervals represented by the thick slices of the sequences. Let p be a pattern and s be a scope, the property p s has the following meaning:

- p globally: the pattern p must hold on the whole execution,
- p before Q: the pattern p must hold before a state property Q occurs,
- p after Q: the pattern p must hold after a state property Q occurs,
- p between Q and R: the pattern p must hold within the system execution intervals from an occurrence of Q to the next occurrence of R,
- p after Q unless R has the same meaning of p between Q and R, but it must hold even if the state property R does not occur.

It is clear that the patterns of DAC dramatically simplify the specification of temporal properties, with a fairly complete coverage. Indeed, they collected hundreds of specifications and they observed that 92% of them fall into this small set of patterns/scopes [4]. Furthermore, DAC adopt translational semantics and provide a complete library [3], mapping each pattern/scope combination to the corresponding formula in many formalisms (e.g. LTL, CTL, Quantified Regular Expressions, μ-calculus). For example, for each scope s, this library maps the property schema P′ responds to P s to the equivalent LTL formula as it is given in Table 1.

Table 1. DAC's LTL Mappings of P′ responds to P s

Scope s	LTL
globally	$\Box(P \Rightarrow \Diamond P')$
before Q	$\Diamond Q \Rightarrow (P \Rightarrow (\neg Q \ \mathsf{U}\ (P' \wedge \neg Q)))\ \mathsf{U}\ Q$
after Q	$\Box(Q \Rightarrow \Box(P \Rightarrow \Diamond P'))$
between Q and R	$\Box((Q \wedge \neg R \wedge \Diamond R) \Rightarrow (P \Rightarrow (\neg R\ \mathsf{U}\ (P' \wedge \neg R)))\ \mathsf{U}\ R)$
after Q unless R	$\Box(Q \wedge \neg R \Rightarrow ((P \Rightarrow (\neg R\ \mathsf{U}\ (P' \wedge \neg R)))\ \mathsf{W}\ R)$

We may note that DAC define informally the generic patterns: bounded existence [eventually P at most k times], chain precedence [$(\mathsf{P}_1, \ldots, \mathsf{P}_n)$ precedes $(\mathsf{P}'_1, \ldots, \mathsf{P}'_m)$] and chain response [$(\mathsf{P}'_1, \ldots, \mathsf{P}'_m)$ responds to $(\mathsf{P}_1, \ldots, \mathsf{P}_n)$]. But because of the translational semantics they can only consider and translate a limited number of cases that are the ten patterns listed above.

3 Compositional Automata-based Semantics

In our approach, the semantics of the temporal properties is defined compositionally by automata composition. Any pattern p is defined by a Büchi automaton pa where the transitions are labeled by state propositions. Any scope s is defined by a specialized Büchi automaton that has a special state, called *composition state* and noted cs, in which a pattern automaton pa can be replaced. Hence, the resulting automaton corresponding to a property $\mathsf{p}\ \mathsf{s}$ is defined by substituting the composition state cs of the scope automaton sa by a pattern automaton pa. The resulting automaton is then a Büchi automaton that accepts all the infinite executions (or *runs*) that satisfy the property.

3.1 Pattern and Scope Automata

Let $\mathcal{P}$ be a finite set of state propositions. A *Büchi* automaton over $\mathcal{P}$ is a finite-state automaton which accepts infinite words. It is formally defined by a 5-tuple $(Q, init, F, \mathcal{P}, T))$ where Q is a finite set of states, $init(\in Q)$ is the initial state, $F(\subseteq Q)$ is a set of accepting states and $T(\subseteq Q \times \mathcal{P} \times Q)$ is a labeled transition relation.

An infinite word $P_1\ P_2\ \dots\ P_n\ \dots$ is accepted by a *Büchi* automaton if there exists a run $q_0 \xrightarrow{P_1} q_1 \xrightarrow{P_2} q_2 \dots q_{n-1} \xrightarrow{P_n} q_n \dots$ such that $q_0 = init$ and each step of the run is a transition ($\forall i \in \mathbb{N},\ q_i \xrightarrow{P_{i+1}} q_{i+1} \in T$) and the set of accepting states within the run is infinite ($\{i \in \mathbb{N} \mid q_i \in F\}$).

While a pattern is described as a Büchi automaton, a scope s is a Büchi automaton which has a composition state cs representing a generic pattern. Hence, a temporal property $\mathsf{p}\ \mathsf{s}$ is described by a standard Büchi automaton that is the scope one in which the composition state is substituted by the pattern automaton.

Definition 1 (Pattern and Scope Automata). *Let $\mathcal{P}$ be a finite set of state propositions. A* ***pattern automaton*** *is defined by a Büchi automaton* $\mathsf{pa} \overset{def}{=} (Q_{\mathsf{pa}}, init_{\mathsf{pa}}, F_{\mathsf{pa}}, \mathcal{P}, T_{\mathsf{pa}})$ *and a* ***scope automaton*** *by a Büchi automaton* $\mathsf{sa} \overset{def}{=} (Q_{\mathsf{sa}} \cup \{\mathsf{cs}\}, init_{\mathsf{sa}}, F_{\mathsf{sa}}, \mathcal{P}, T_{\mathsf{sa}})$ *in which the set of states is the disjoint union of a set of standard states* Q_{sa} *and a* ***composition state*** *denoted* cs.

Figure 2 illustrates the pattern automata associated to the patterns presented in Sec. 2. The initial states are pointed to by incoming arrows while the accepting states are marked by double circles. We give here the complements of both chain response patterns because the complements are simpler and smaller (3 states instead of 6). The reader may know that Büchi automata are closed under complementation and there are many construction algorithms [9]. In Sec. 4, we will explain how we proceed to automatically obtain all these pattern automata.

Fig. 3 illustrates the scope automata associated with the scopes presented in Sec. 2. Squares are used to represent the composition states. Double squares are accepting composition states. In Sec. 4, we will explain how we proceed to automatically obtain all these scope automata.

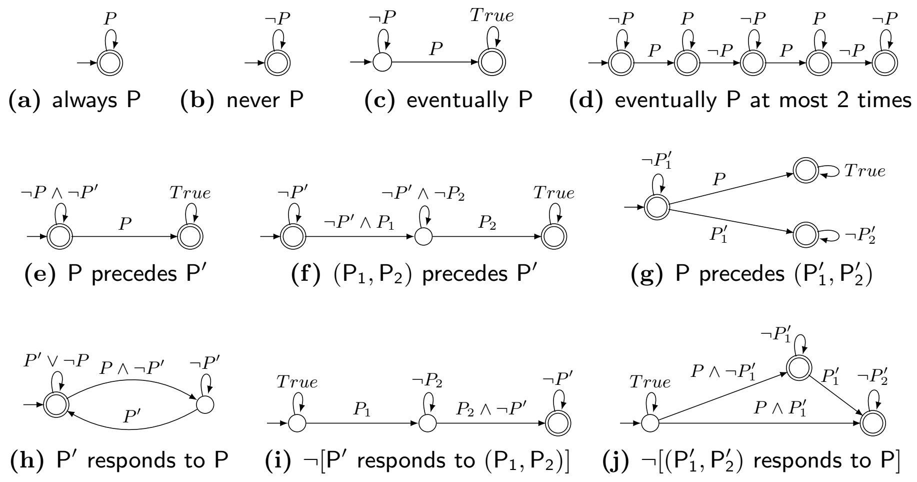

(a) always P
(b) never P
(c) eventually P
(d) eventually P at most 2 times
(e) P precedes P′
(f) (P1, P2) precedes P′
(g) P precedes (P′1, P′2)
(h) P′ responds to P
(i) ¬[P′ responds to (P1, P2)]
(j) ¬[(P′1, P′2) responds to P]

Fig. 2. Pattern Automata

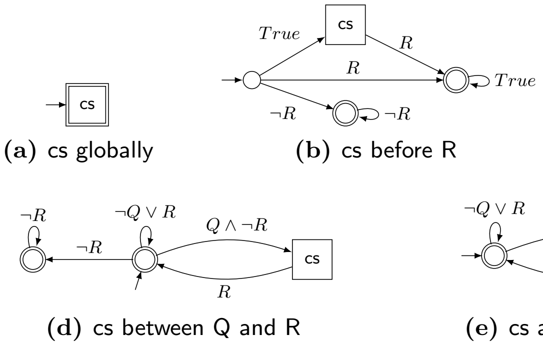


Fig. 3. Scope Automata

3.2 Composition

In this subsection, we formally define the operation of substitution of the composition state cs by a pattern automaton pa in a scope automaton sa.

Definition 2 (Composition Operation). *Let* pa $\overset{def}{=} (Q_{\mathsf{pa}}, init_{\mathsf{pa}}, F_{\mathsf{pa}}, \mathcal{P}, T_{\mathsf{pa}})$ *be a pattern automaton and* sa $\overset{def}{=} (Q_{\mathsf{sa}} \cup \{\mathsf{cs}\}, init_{\mathsf{sa}}, F_{\mathsf{sa}}, \mathcal{P}, T_{\mathsf{sa}})$ *be a scope automaton where* cs *is the composition state of* sa. *The* ***substitution of the state*** cs ***by*** pa ***in*** sa *is the Büchi automaton* $(Q, init, F, \mathcal{P}, T)$ *where:*

– $Q = Q_{\mathsf{pa}} \cup Q_{\mathsf{sa}}$

– $init \overset{def}{=} \begin{cases} init_{\mathsf{sa}} & \mathtt{if}\ init_{\mathsf{sa}} \neq \mathsf{cs} \\ init_{\mathsf{pa}} & \mathtt{otherwise} \end{cases}$ – $F \overset{def}{=} \begin{cases} F_{\mathsf{sa}} & \mathtt{if}\ \mathsf{cs} \notin F_{\mathsf{sa}} \\ (F_{\mathsf{sa}} \setminus \{\mathsf{cs}\}) \cup F_{\mathsf{pa}} & \mathtt{otherwise} \end{cases}$

– *$T \subseteq Q \times \mathcal{P} \times Q$ is the smallest relation defined by the following rules:*

1. *Pattern transitions:* $$\frac{q \xrightarrow{P} q' \in T_{\mathsf{pa}}}{q \xrightarrow{P \wedge R} q' \in T}$$

where $R \stackrel{def}{=} \bigwedge_{P' \in Out(\mathsf{cs})} \neg P'$ *and* $Out(\mathsf{cs}) = \{P' \mid \exists q''.(q'' \in Q_{\mathsf{sa}} \wedge \mathsf{cs} \xrightarrow{P'} q'' \in T_{\mathsf{sa}})\}$

2. *Left-closed scope opening transitions:* $$\frac{q \xrightarrow{P} \mathsf{cs} \in T_{\mathsf{sa}},\ init_{\mathsf{pa}} \xrightarrow{P'} q' \in T_{\mathsf{pa}}}{q \xrightarrow{P \wedge P'} q' \in T}$$

3. *Right-open scope closing transitions:* $$\frac{\mathsf{cs} \xrightarrow{P} q' \in T_{\mathsf{sa}},\ q \in F_{\mathsf{pa}}}{q \xrightarrow{P} q' \in T}$$

4. *Other scope transitions:* $$\frac{q \xrightarrow{P} q' \in T_{\mathsf{sa}},\ q, q' \in Q_{\mathsf{sa}}}{q \xrightarrow{P} q' \in T}$$

The resulting set of states is the union of the sets of states without the composition state cs. The initial state is the initial state of the pattern if the composition state is initial, otherwise it is the initial state of the scope. When the composition state cs is an accepting one, the set of accepting states is the union of both sets of accepting states without cs. Otherwise, it is only composed of those of the scope.

The resulting transitions are defined as follows. The rule 1 adds each transition within the pattern automaton after modifying the label into $P \wedge \neg P'_0 \wedge \cdots \wedge \neg P'_n$ where $P'_i,\ i \in [0, \ldots, n]$ are the labels carried by the n transitions outgoing from the composition state cs as illustrated in Fig. 4(a) where the rectangle represents the composition state cs having two outgoing transitions. This restriction of the labels on the pattern transitions is applied in order to avoid that they capture the scope ones, hence scope transitions keep priority. For example, any transition of the pattern P′ responds to P does not satisfy the condition R which is the exit condition of the scope between Q and R. Indeed, the condition R must not be satisfied (i.e. the exit of the scope must not be possible) before reaching the pattern's accepting state where outgoing transitions hold R (see rule 3). The rule 2 synchronizes the transitions of the scope leading to the composition state cs with the initial transitions of the pattern by making the conjunction of their labels as it is illustrated in Fig. 4(b). For every transition outgoing from the composition state cs, the rule 3 adds a transition from every accepting state of the pattern as illustrated in Fig. 4(c). Rule 2 makes the scope interval left-closed and rule 3 makes it right-open, this aspect will be detailed in Sec. 5. Finally, the rule 4 adds each transition of the scope automaton in which the composition state cs is not involved.

Example 1 (Composition of Automata). Fig. 5 shows the Büchi automaton obtained by applying the composition operation given in Def. 2 to the temporal property P′ responds to P between Q and R.

Our composition operation is made in a linear complexity w.r.t. the size of the pattern and scope automata. Thus, this automata-based approach yields a technique to transform each DAC temporal property into a Büchi automaton from two Büchi automata in a linear complexity. In contrast, building the same Büchi

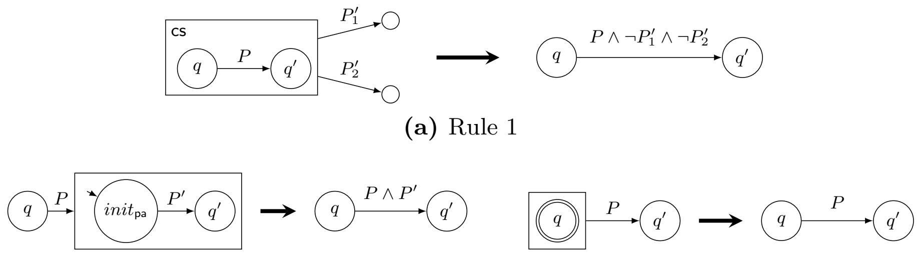


Fig. 4. Illustration of Composition Rules

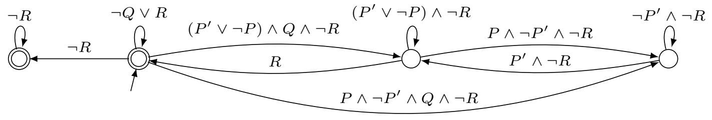


Fig. 5. P′ responds to P between Q and R

automaton from the LTL formula given as translation would be exponential w.r.t. the size of the formula [7].

4 Comparison of Both Semantics

In this section, we present the experiments we conducted in order to measure the consistency of our compositional semantics against the translational semantics given by DAC [3]. We do so by comparing our resulting automata with the LTL formulæ given by DAC.

For these experiments, we used the GOAL (Graphical Tool for Omega-Automata and Logics) tool [11] that is an adequate graphical tool for defining and manipulating Büchi automata and temporal logic formulæ. GOAL supports the translation of temporal formulæ such as Quantified Propositional Temporal Logic (QPTL) into Büchi automata where many well-known translation algorithms (e.g. LTL2BA [6]) are implemented and most of them support past operators. It also provides language equivalence between two Büchi automata thanks to efficient complementation, intersection and emptiness algorithms. As the recent implementation of GOAL is based on the Java Plugin Framework, it can be properly extended by new plugins, providing new functionalities that are loaded at run-time. We implemented our composition algorithm within an independent plug-in that we make available at the web page [10].

The process we applied to do our experiments can be summarized as follows:

1. We used the DAC's LTL formulæ that correspond to the properties combining any pattern with the globally scope to generate the patterns automata using GOAL. These formulæ are shown in Table 2 within the globally column.

Note that, using our composition operation, the substitution of some pattern p within the scope globally keeps unchanged the automaton pa. This is the way we obtained the pattern automata previously presented in Fig. 2.

2. We used the DAC's LTL formulæ that correspond to the properties combining the always P pattern with any scope to generate the scope automata using GOAL. These formulæ are shown in Table 2 within the always P row. Interpreting the unique state having the P loop transition as the composition state, we obtained the scope automata previously presented in Fig. 3.
3. We ran our composition to automatically generate the automata for all pattern/scope combinations. We compared them with the automata obtained directly from the corresponding DAC's LTL formulæ given in [3]. The results of the comparison are given in Table 2. For each combination, the automaton of the translational semantics may be equivalent ($\equiv$), strictly included ($\subset$), strictly superior ($\supset$) or not included nor superior ($\neq$) to the automaton given by composition. The non-equivalent cases are indexed by a case number which we use below to explain the reasons behind the mismatching.

Table 2. Comparison between DAC's Semantics and Compositional Semantics

	globally	before R	after Q	between Q and R	after Q unless R
always P	$\Box P$	$\Diamond R \Rightarrow (\neg P \ \mathsf{U}\ R)$	$\Box(\neg Q) \vee \Diamond(Q \wedge \Diamond P)$	$\Box(Q \wedge \neg R \Rightarrow (\neg R \ \mathsf{W}\ (P \wedge \neg R)))$	$\Box(Q \wedge \neg R \Rightarrow (\neg R \ \mathsf{U}\ (P \wedge \neg R)))$
never P	$\Box \neg P$	$\equiv$	$\equiv$	$\equiv$	$\equiv$
eventually P	$\Diamond P$	$\subset$ (1)	$\equiv$	$\subset$(2)	$\subset$ (2)
eventually P at most 2 times	$(\neg P \ \mathsf{W}\ (P \ \mathsf{W}\ (\neg P \ \mathsf{W}\ (P \ \mathsf{W}\ \Box\neg P))))$	$\equiv$	$\equiv$	$\equiv$	$\equiv$
P precedes P′	$\neg P' \ \mathsf{W}\ P$	$\equiv$	$\supset$ (3)	$\subset$ (2)	$\subset$ (2)
(P$_1$, P$_2$) precedes P′	$\Diamond P' \Rightarrow (\neg P' \ \mathsf{U}\ (P_1 \wedge \neg P' \wedge \bigcirc(\neg P' \ \mathsf{U}\ P_2)))$	$\equiv$	$\equiv$	$\subset$ (2)	$\subset$ (2)
P precedes (P′$_1$, P′$_2$)	$(\Diamond(P'_1 \wedge \bigcirc\Diamond P'_2)) \Rightarrow ((\neg P'_1) \ \mathsf{U}\ P))$	$\equiv$	$\equiv$	$\subset$ (2)	$\subset$ (2)
P′ responds to P	$\Box(P \Rightarrow \Diamond P')$	$\equiv$	$\equiv$	$\equiv$	$\equiv$
(P′$_1$, P′$_2$) responds to P	$\Box(P \Rightarrow \Diamond(P'_1 \wedge \bigcirc\Diamond P'_2))$	$\supset$ (4)	$\equiv$	$\supset$ (4)	$\supset$ (4)
P′ responds to (P$_1$, P$_2$)	$\Box(P_1 \wedge \bigcirc\Diamond P_2 \Rightarrow \bigcirc(\Diamond(P_2 \wedge \Diamond P')))$	$\neq$ (5)	$\equiv$	$\neq$ (5)	$\neq$ (5)

Table 3 provides for each mismatching case the formula proposed by DAC and the corresponding formula (verified using GOAL) to the composition automaton we obtained by our composition algorithm. We call this formula *composition formula* and we underline the differences. We use the symbol $\ominus$ for "previous" (resp. B for "back-to") to represent the past-time dual operator of the future operator $\bigcirc$ for "next" (resp., W for "weak-until"). We use the past temporal operators only in case (2) to obtain a concise formula. The reader may know that past-time modalities do not add expressive power to future linear-time temporal logic but it can be exponentially more succinct [8].

The mismatching case (1) emphasizes that DAC's formula does not recognize the case where R occurs at the initial state, so the interval of the scope is empty since the interval is right-open (see details in Sec. 5) and the

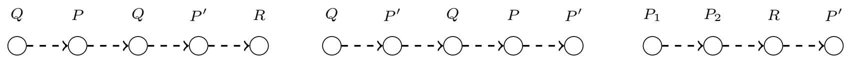
Table 3. Mismatching Cases

Mismatching cases	**DAC's Formula**		**Composition Formula**
(1) eventually P before R	$\neg R \mathsf{W} (P \wedge \neg R)$	$\subset$	$\underline{R \ \vee \ }\neg R \mathsf{W} (P \wedge \neg R)$
(2) eventually P/Precedence between Q and R/after Q unless R	$\Box((Q \wedge \neg R) \Rightarrow \ldots)$	$\subset$	$\Box((Q \ \underline{\wedge \ominus (\neg Q \ \mathsf{B} \ R)} \wedge \neg R) \Rightarrow \ldots)$
(3) P precedes P$'$ after Q	$\Box\neg Q \vee \Diamond(Q \wedge (\neg P' \ \mathsf{W} \ P))$	$\supset$	$\Box\neg Q \vee (\underline{\neg Q \ \mathsf{U}\ }(Q \wedge (\neg P' \ \mathsf{W} \ P)))$
(4) $(\mathsf{P}'_1, \mathsf{P}'_2)$ responds to P before R/ . . .	$\ldots(P \Rightarrow (\neg R \ \mathsf{U} \ (P'_1 \wedge \neg R \wedge \bigcirc (\neg R \ \mathsf{U} \ P'_2)))) \ldots$	$\supset$	$\ldots(P \Rightarrow (\neg R \ \mathsf{U} \ (P'_1 \wedge \neg R \wedge \bigcirc (\neg R \ \mathsf{U} \ (P'_2\underline{\wedge\neg R}))))) \ldots$
(5) P$'$ responds to $(\mathsf{P}_1, \mathsf{P}_2)$ before R/ . . .	$\ldots(P_1 \wedge \bigcirc(\neg R \ \mathsf{U} \ P_2) \Rightarrow \bigcirc(\neg R \ \mathsf{U} \ (P_2 \wedge \Diamond P'))) \ldots$	$\neq$	$\ldots(P_1 \wedge \bigcirc(\neg R \ \mathsf{U} \ (P_2\underline{\wedge\neg R})) \Rightarrow \bigcirc(\neg R \ \mathsf{U} \ (P_2 \wedge \underline{(\neg R \ \mathsf{U} \ (P' \wedge \neg R))}))) \ldots$

property eventually P before R is obviously true. It was an oversight as all other LTL formulæ of the before scope handle such empty interval cases.

Considering case (2), DAC mention in the notes published in [3] that the **first** occurrence of Q opens the intervals of the scopes after Q, between Q and R and after Q unless R (See the right part of Fig. 1). However, some of the proposed formulæ are unfaithful to the **first** occurrence semantics as they consider **all** occurrences of Q. For example, the trace of Fig. 6a that verifies the property P precedes P$'$ between (first) Q and R, is accepted by our generated composition automaton (equivalent to $\Box((Q \wedge \neg R \ \underline{\wedge \ominus (\neg Q \ \mathsf{B} \ R)} \wedge \Diamond R) \Rightarrow (\neg P \ \mathsf{U} \ (P' \vee R))))$, but it is rejected by the formula given by DAC (i.e. $\Box((Q \wedge \neg R \wedge \Diamond R) \Rightarrow (\neg P \ \mathsf{U} \ (P' \vee R))))$. The past predicate $\underline{\ominus(\neg Q \ \mathsf{B} \ R)}$ (i.e. previous ($\neg Q$ back-to R)) ensures that only the first occurrence of Q is considered as there is no occurrence of Q in the past since the last occurrence of R if there is any. We note here that expressing with future modalities the first occurrence of Q following some occurrence of R in between Q and R or after Q unless R, is tedious. For example, the equivalent pure future formula of P precedes P$'$ between (first) Q and R is:

$$(\neg(Q \wedge \neg R \wedge \Diamond R) \ \mathsf{W} \ ((Q \wedge \neg R \wedge \Diamond R) \wedge (\neg P' \ \mathsf{W} \ (P \vee R))))$$
$$\underline{\wedge \ \Box(R \Rightarrow (\neg(Q \wedge \neg R \wedge \Diamond R) \ \mathsf{W} \ ((Q \wedge \neg R \wedge \Diamond R) \wedge}\ (\neg P' \ \mathsf{W} \ (P \vee R))))).$$



(a) case (2) **(b)** case (3) **(c)** case (5)

Fig. 6. Mismatching examples

In case (3), the formulæ proposed by DAC consider **any** occurrence of Q ($\Diamond(Q \wedge \ldots)$) rather than the **first** occurrence $(\underline{\neg Q \ \mathsf{U} \ }(Q \wedge \ldots))$. As a typical example, the trace of Fig. 6b that does not verifiy the property P precedes P$'$ after Q, is rejected by our generated composition automaton but it is accepted by the formula given by DAC.

Cases (2) and (3) are quite similar and the question "why such a mismatching does not happen for other patterns?" obviously arises. There are two answers depending on cases. In most cases, DAC handle the **first** occurrence semantics within their LTL formulæ, in other cases, patterns are response-oriented properties where the **all** and **first** occurrence of Q semantics are equivalent. For example the LTL formula given by DAC of the property P' responds to P after Q, i.e. $\Box(Q \Rightarrow \Box(P \Rightarrow \Diamond P'))$ (**all** occurrences) is equivalent to ours $\underline{\Diamond Q \Rightarrow (\neg Q \ \mathsf{U}\ (Q \ \wedge \ \Box(P \Rightarrow \Diamond P'}))$ (**first** occurrence).

DAC have chosen to define scopes as right-open intervals that do not include the state marking the end of the scope [3]. In case (4) and a part of case (5), DAC provide formulæ where P_2' and R can occur simultaneously; that is unfaithful to the right-open scope semantics.

In case (5), the DAC formula of P' responds to $(\mathsf{P}_1, \mathsf{P}_2)$ using the modality eventually $(\Diamond R)$ does not require that the response P' occurs within the scope! (i.e. before R). For example, the trace of Fig. 6c that does not verify the property P' responds to $(\mathsf{P}_1, \mathsf{P}_2)$ before R, is rejected by our generated composition automaton but it is accepted by the formula given by DAC.

The experiments, we did here, show the homogeneity of our composition semantics and reveal many different interpretations of the same scope within the translational semantics given by DAC. This emphasizes that it is difficult to give faithful LTL translation to all combinations of patterns and scopes. Our composition semantics brings a valuable consistency.

5 Genericity and Extensibility of the Approach

In the following, we will show the genericity and extensibility of our composition semantics. First we propose some generic patterns and some variant scopes that are not supported by the translational semantics of DAC and then we show their corresponding representation using our automata-based approach.

5.1 Generic Patterns and Variants of Scopes

First, we consider generic patterns that DAC have defined informally but they do not succeed to translate them into equivalent generic temporal logic formulas, hence they only translated a limited number of obvious cases.

- In DAC's work, the pattern eventually has only two forms: eventually P means that P is true at least once and eventually P at most 2 times means that the states switch from ¬P to P at most twice (see Fig. 2). We consider three new generic variants to this pattern: eventually P k times, eventually P at least k times and eventually P at most k times that mean respectively: the state P *becomes true* exactly k times, at least k times and at most k times where k is some natural integer constant.
- Similarly, we propose three generic variants of the eventually pattern considering the number of all occurrences of state P rather than the number of switching occurrences from ¬P to P. We call them precisely P k times, precisely P at least k times and precisely P at most k times.
- Finally, we consider both Chain Precedence and Chain Response patterns having the generic forms $[(\mathsf{P}_1, \ldots, \mathsf{P}_n)$ precedes $(\mathsf{P}'_1, \ldots, \mathsf{P}'_m)]$ and $[(\mathsf{P}'_1, \ldots, \mathsf{P}'_m)$ responds to $(\mathsf{P}_1, \ldots, \mathsf{P}_n)]$.

Next, we propose some enhancements to improve the expressiveness of scopes. These enhancements are inspired by the DAC's notes [3] and our needs within the TASCCC project [1].

– DAC have chosen to define scopes as right-open intervals (i.e. left-closed) that include the state marking the beginning of the scope, but do not include the state marking the end of the scope. We extend scopes with support to open the scope on the left or close it on the right. Hence, we add one variant for both the before _ and after _ scopes and three supplementary variants for the between _ and _ and after _ unless _ scopes. We chose DAC's semantics as the default semantics.
– In DAC's work, between Q and R and after Q unless R scopes are interpreted relatively to the **first** occurrence of Q (see Fig. 1). We keep the first occurrence as default semantics and we add variants to support the **last** occurrence semantics.

The syntax of our extended pattern-based language is summarized in Fig. 7. Non-terminals are indicated by *italics*, keywords are in policy and terminals are underlined. For example $\underline{\text{a}}$ is an atomic proposition. $(\ldots)?$ designates an optional part. The element P stands for a state property which is a boolean proposition over the alphabet of the different atomic propositions and the optional element '[' or ']' stands for the interval's nature, open or closed at each endpoint.

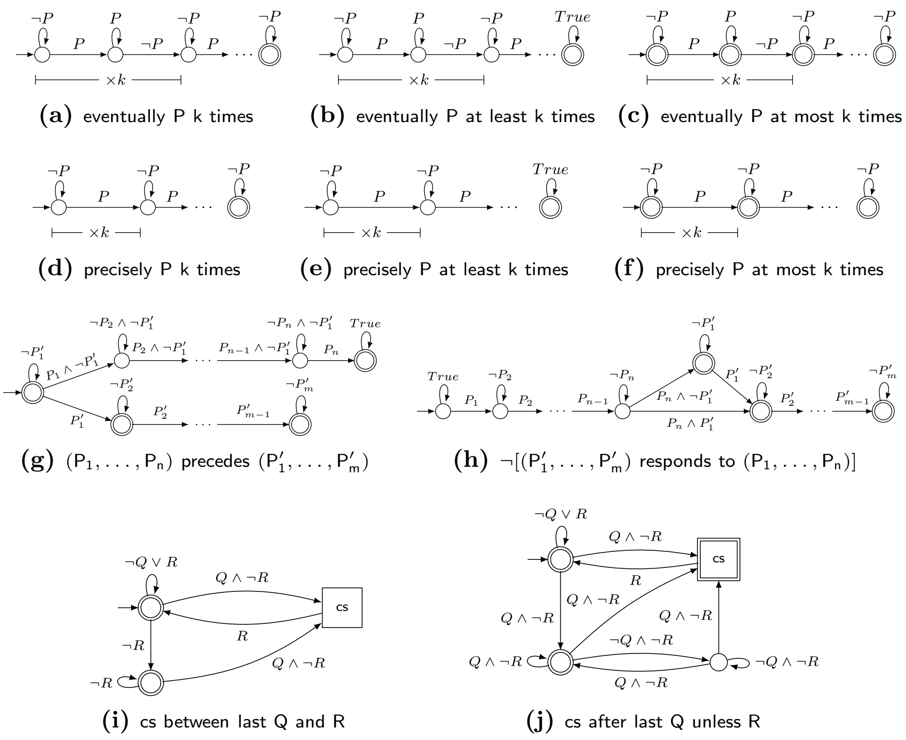
$$
\begin{array}{lll}
\textit{Property} & ::= & \textit{Pattern Scope} \\
\textit{Pattern} & ::= & \mathsf{always}\ P \\
 & \mid & \mathsf{never}\ P \\
 & \mid & (\mathsf{eventually} \mid \mathsf{precisely})\ P\ ((\mathsf{at\ least} \mid \mathsf{at\ most})?\ \underline{\text{integer}}\ \mathsf{times})? \\
 & \mid & \textit{Chain}\ \mathsf{precedes}\ \textit{Chain} \\
 & \mid & \textit{Chain}\ \mathsf{responds\ to}\ \textit{Chain} \\
\textit{Scope} & ::= & \mathsf{globally} \\
 & \mid & \mathsf{before}\ P\ (\text{`['} \mid \text{`]'})? \\
 & \mid & \mathsf{after}\ (\text{`['} \mid \text{`]'})?\ P \\
 & \mid & \mathsf{between}\ (\text{`['} \mid \text{`]'})?\ \mathsf{last}?\ P\ \mathsf{and}\ P\ (\text{`['} \mid \text{`]'})? \\
 & \mid & \mathsf{after}\ (\text{`['} \mid \text{`]'})?\ \mathsf{last}?\ P\ \mathsf{unless}\ P\ (\text{`['} \mid \text{`]'})? \\
\textit{Chain} & ::= & P \mid P\ \text{`,'}\ \textit{Chain} \\
P & ::= & \underline{\text{a}} \mid \mathsf{true} \mid \neg P \mid P \vee P
\end{array}
$$

Fig. 7. Syntax of Enriched DAC's Temporal Properties

5.2 Variant Semantics

The generic patterns and the **last** variants of scopes such as between last Q and R are directly expressed in our approach by describing their suitable automata, since their semantics does not have impact on the composition definition (Def. 2) given in Sec. 3. Fig. 8 shows graphically their associated automata.

However, the scopes variants on the closure and opening of the intervals such as before Q] or after] Q require some generalizations in the composition definition. Indeed, in Def. 2, we did not make a distinction between right-open and


(a) eventually P k times (b) eventually P at least k times (c) eventually P at most k times

(d) precisely P k times (e) precisely P at least k times (f) precisely P at most k times

(g) $(P_1, \ldots, P_n)$ precedes $(P'_1, \ldots, P'_m)$

(h) $\neg[(P'_1, \ldots, P'_m)$ responds to $(P_1, \ldots, P_n)]$

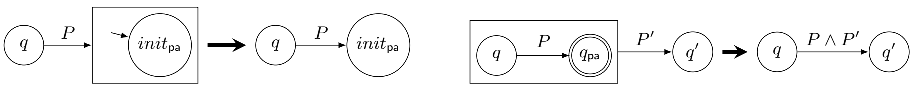
(i) cs between last Q and R

(j) cs after last Q unless R

Fig. 8. Automata of Generic Pattern and Scope Variants

right-closed intervals, and left-open and left-closed intervals. We have chosen by default the right-open and left-closed intervals as initially given by DAC [4,3]. An interval left/right-open corresponds to a strict composition while a left/right-closed corresponds to a non-strict composition. A strict composition means that given a composition state cs, its ingoing transitions should be completely executed before the transitions outgoing from the initial state of the pattern automaton are triggered (left-open, see Fig. 9a), and the transitions ingoing in the accepting states of the pattern automaton should be completely performed before the outgoing transitions of cs are triggered (right-open, see Fig. 4c). A non-strict substitution means that the transitions outgoing from the initial state of the pattern automaton should be simultaneously executed with the ingoing transitions of cs (left-closed, see Fig. 4b), and the transitions ingoing in the accepting states of the pattern automaton should be simultaneously executed with the outgoing transitions of cs (right-closed, see Fig. 9b).

Hence, to describe sequencing relationships between the states of the scope automaton and the pattern automaton at the left and the right borders of the composition state, we add the following rules $2'$ and $3'$ to the composition definition Def. 2 :



(a) left-open **(b)** right-closed

Fig. 9. Illustration of Left-open and Right-closed Composition Rules

2'. Left-open scope opening transitions: $\dfrac{q \xrightarrow{P} \mathsf{cs} \in T_{\mathsf{sa}}}{q \xrightarrow{P} init_{\mathsf{pa}} \in T}$

3'. Right-closed scope closing transitions: $\dfrac{q \xrightarrow{P} q_{\mathsf{pa}} \in T_{\mathsf{pa}},\ q_{\mathsf{pa}} \in F_{\mathsf{pa}},\ \mathsf{cs} \xrightarrow{P'} q' \in T_{\mathsf{sa}}}{q \xrightarrow{P \wedge P'} q' \in T}$

Due to the compositional semantics we adopted to support the patterns and the scopes proposed by DAC, our language is generic and easily extensible. To add a new variant of any pattern or any scope, it suffices to describe it once in terms of an automaton. This is much easier than specifying all resulting combinations in LTL. We only need to specify the n patterns plus the m scopes to generate the $n \times m$ combinations. In our extension for [1], we have identified above 12 patterns and 21 scopes. To describe them using DAC's semantics, we need to translate 252 combinations whereas following our approach, it suffices to specify the 17 scheme of automata of Fig. 2, Fig. 3 and Fig. 8.

6 Conclusion and Future Work

In this paper, we present a compositional semantics of the DAC's property language. It is defined by the automata of the patterns and the scopes and by the composition operation. We compare it with DAC's translational semantics associating an LTL formula with each pattern/scope combination. This comparison emphasizes the homogeneity of our semantics and reveals that the interpretations of many scopes within their semantics are unfaithful w.r.t the informal definitions given in [4,5].

Our semantics being compositional, the property language is generic and easily extensible. In this paper, we have shown that handling generic patterns and adding new scope variants, only require to give their semantics by automata. Then, the composition operation gives the semantics of all properties that can be described by combining any new pattern with all existing scopes and combining any new scope with all existing patterns. We also made explicit both the closing and opening default choices of the DAC's semantics by generalizing the composition operation. Moreover, our approach, consisting to choose a scope and a pattern automata, is more efficient for automata-based verification of properties or coverage evaluation of test sequences than a method which consists to choose an LTL formula because it replaces the exponential LTL formula translation into automata by a linear automata composition.

In [2], we are currently using this approach for the evaluation of the coverage of dynamic properties (described as a pattern and a scope composition) by a test suite. The works that we present here are limited to combine one pattern with one scope. We aim to generalize this work by combining several patterns with a succession of scopes.

References

1. Cabrera Castillos, K., Dadeau, F., Julliand, J., Taha, S.: Projet TASCCC, Test Automatique basé sur des SCénarios et évaluation Critères Communs., `http://lifc.univ-fcomte.fr/TASCCC/`
2. Cabrera Castillos, K., Dadeau, F., Julliand, J., Taha, S.: Measuring test properties coverage for evaluating UML/OCL model-based tests. In: Wolff, B., Zaïdi, F. (eds.) ICTSS 2011. LNCS, vol. 7019, pp. 32–47. Springer, Heidelberg (2011)
3. Dwyer, M.B., Alavi, H., Avrunin, G., Corbett, J., Dillon, L., Pasareanu, C.: Specification Patterns, `http://patterns.projects.cis.ksu.edu/`
4. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st International Conference on Software Engineering, pp. 411–420 (1999)
5. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: FMSP, pp. 7–15 (1998)
6. 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)
7. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV, pp. 3–18. Chapman, Hall, Ltd., London (1996)
8. Markey, N.: Temporal logic with past is exponentially more succinct, concurrency column. Bulletin of the EATCS 79, 122–128 (2003)
9. Sistla, A.P., Vardi, M.Y., Wolper, P.: The complementation problem for Büchi automata with applications to temporal logic. Theoretical Computer Science 49(2-3), 217–237 (1987)
10. Taha, S.: OCL temporal extension (2012), `http://wwwdi.supelec.fr/taha/temporalocl/`
11. Tsay, Y.K., et al.: Graphical Tool for Omega-Automata and Logics, `http://goal.im.ntu.edu.tw/wiki/doku.php`

A Formal Semantics for Complete UML State Machines with Communications*

Shuang Liu[1], Yang Liu[2], Étienne André[3], Christine Choppy[3], Jun Sun[4], Bimlesh Wadhwa[1], and Jin Song Dong[1]

[1] School of Computing, National University of Singapore, Singapore
[2] Nanyang Technology University, Singapore
[3] Université Paris 13, Sorbonne Paris Cité, LIPN, F-93430, Villetaneuse, France
[4] Singapore University of Design and Technology, Singapore

Abstract. UML is a widely used notation, and formalizing its semantics is an important issue. Here, we concentrate on formalizing UML state machines, used to express the dynamic behaviour of software systems. We propose a formal operational semantics covering all features of the latest version (2.4.1) of UML state machines specification. We use labelled transition systems as the semantic model, so as to use automatic verification techniques like model checking. Furthermore, our proposed semantics includes synchronous and asynchronous communications between state machines. We implement our approach in USM^{2}C, a model checker supporting editing, simulation and automatic verification of UML state machines. Experiments show the effectiveness of our approach.

1 Introduction

UML state machines are widely used to model the dynamic behaviour of an object. Since the UML specification is documented in natural language, inconsistencies and ambiguities arise, and it is thus important to provide a formal semantics for UML state machines. A formal semantics (1) allows more precise and efficient communication between engineers, (2) yields more consistent and rigorous models, and (3) lastly and most importantly, enables automatic formal analysis of UML state machines.

However, existing works only provide formal semantics for a subset of UML state machines features, leaving some important issues unaddressed. A few approaches [19,22] consider the non-determinism in the presence of orthogonal composite states, which is an important modelling concept. Although extensibility of the syntax structure is important due to the refinement operations on UML state machines, the syntax formats defined in those works does not extend well. A semantics able to support the full set of syntax features will help to bring the expressive power of UML state machines to life.

Secondly, in the existing approaches, the event pool mechanism and the communications between state machines are not thoroughly addressed. UML state machines are used to model the behaviour of objects. The whole system may include several state

* This work is supported by project 9.10.11 "Software Verification from Design to Implementation" of Programme Merlion (official collaborative grant co-funded by France and Singapore).

machines interacting with each other synchronously or asynchronously. Enabling the verification of the entire system is quite important, especially in the presence of synchronous communications, which are more likely to cause deadlock situations.

Lastly, the unclarities (that is, inconsistencies and ambiguities) in the UML state machines specifications are not thoroughly checked and discussed. Fecher et al. [8] discussed 29 unclarities in UML 2.0 state machines. But there are still some unclarities (such as the granularity of a transition execution sequence) that are not covered in [8] but will be discussed in this work.

This work aims at bridging the gaps in the existing approaches with the following contributions. (1) We provide a formal operational semantics for UML 2.4.1 state machines covering the complete set of UML state machines features. In particular, our syntax structure is extensible to state machine refinement and future changes. Our semantics formalization considers non-determinism as well as synchronous and asynchronous communications between state machines. (2) We explicitly discuss the event pool mechanisms and consider deferral events as well as completion events. (3) We report new unclarities in UML 2.4.1 state machines specifications. (4) We develop a self-contained tool USM^{2}C based on the semantics we have defined; it model checks various properties such as deadlock-freeness and linear temporal logic (LTL) properties. We conduct experiments on our tool and results show its effectiveness.

The rest of this paper is organized as follows. Section 2 provides the preliminaries of UML state machines. Section 3 and Section 4 define the syntax and semantics for UML state machines, respectively. Section 5 provides the implementation and evaluation results. Related work is discussed in Section 6. Section 7 addresses the limitations of our work, and concludes the paper with future works.

2 UML State Machines Features and Our Assumptions

2.1 Introduction of Basic Features of UML State Machines

We briefly introduce basic features of UML state machines in this section. We use the *RailCar* system in Fig. 1 (a modified version of the example used in [10]) as a running example. The *RailCar* system is composed of 3 state machines: *Car*, *Handler* and *DepartureSM* (referenced by the *Departure* submachine state in the *Car* state machine). They communicate with each other through synchronous event calls.

Vertices and Transitions. A vertex is a node, which refers to a state, a pseudostate, a final state or a connection point reference. A transition is a relation between a source vertex and a target vertex. It may have a guard, a trigger and an effect. The container of a transition is the region which owns the transition. A compound transition is composed of multiple transitions joined via choice, junction, fork and join pseudostates.

Regions. It is a container of vertices and transitions, and represents the orthogonal parts of a composite state or a state machine. In Fig. 1, the area *[R1]* is a region.

States. There are three kinds of states, viz., simple state (*Idle*), composite state (*Operating*) and submachine state (*Departure*). An orthogonal composite state (*WaitArrivalOK*) has more than one region. States can have optional entry/exit/do behaviours. A do behaviour (*PlaySound* in state *Alerted*) can be interrupted by an event. A state can also

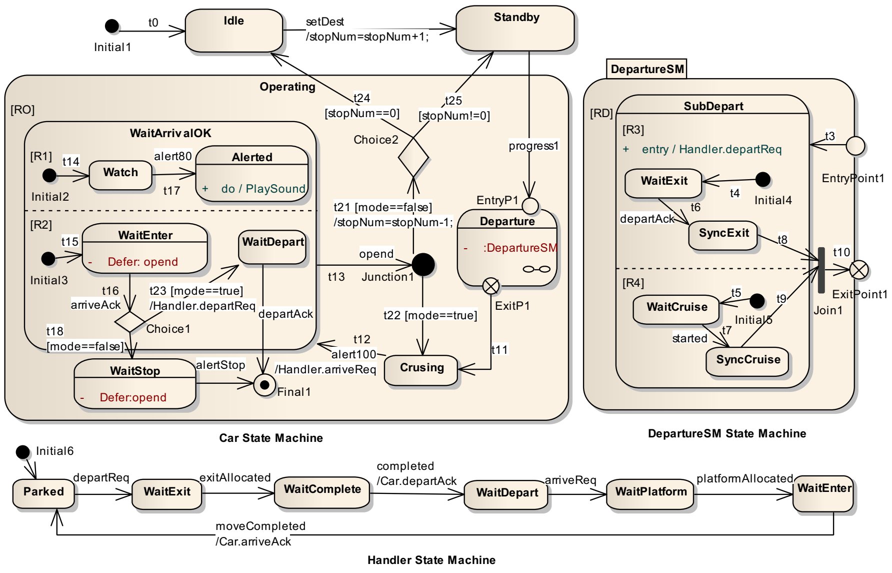


Fig. 1. The RailCar state machine

define a set of deferred events ($\{opend\}$ in state *WaitEnter*). A final state (*Final1*) is a special kind of state which indicates finishing of its enclosing region.

Pseudostates. Pseudostates are introduced to connect multiple transitions to form complex transition paths. There are 10 kinds of pseudostates: initial, join, fork, junction, choice, entry point, exit point, shallow history, deep history, terminate. A join pseudostate (*join1*) is used to merge transitions from states in orthogonal regions. A fork pseudostate is used to split transitions targeting states in orthogonal regions. Junction pseudostates (*Junction1*) represent static branching points. Choice pseudostates (*Choice1*) represent dynamic branching points, i.e., the evaluation of enabled transitions is based on the environment when the choice pseudostate is reached.

Connection Point Reference. It is an entry/exit point of a submachine state and refers to the entry/exit pseudostate of the state machine that the submachine state refers to. In Fig. 1, *EntryP1* and *ExitP1* in *Departure* state are connection point references.

Active State Configuration. It is a set of active states of a state machine when it is in a stable status[1]. In Fig. 1, $\{Operating, Crusing\}$ is an active state configurations.

Run to Completion Step (RTC). It captures the semantics of processing one event occurrence, i.e., executing a set of compound transitions (fired by the event), which may cause the state machine to move to the next active state configuration, accompanied by behaviour executions. It is the basic semantic step in UML state machines. For example in Fig. 1, $\{Operating, WaitArrivalOK, Watch, WaitDepart,\} \xrightarrow{opend} \{Idle\}$ is an RTC step.

[1] The state machine is waiting for event occurrences.

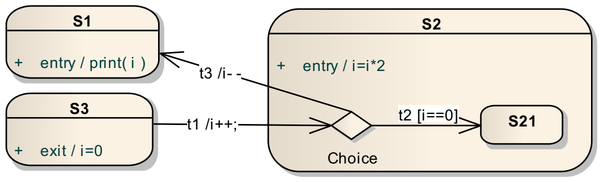


Fig. 2. Illustration of transition execution sequence

2.2 Basic Assumptions on UML State Machines Semantics

We briefly sketch below some new unclarities (detailed in [17]) we found in the UML 2.4.1 state machines specification, as well as our assumptions in this work.

Transition Execution Sequence. Transitions and compound transitions are used in interleaving in the descriptions of transition execution sequence, which raises confusions. The transition execution ordering is important since different execution orders may lead to different results. For example in Fig. 2, Suppose S_3 is active and transition t_1 is fired. If we define the transition execution sequence based on the compound transition, the behaviour execution sequence is "$i = 0;\ i++;\ i--;\ print(i)$ " and 0 should be printed. If we define the transition execution sequence based on a single transition, the behaviour execution sequence should be "$i = 0;\ i++;\ i = i * 2;\ i--;\ print(i)$" and 1 should be printed. In the first case, the entry behaviour of state *S2* is not executed, which contradicts the semantics of entry behaviours. We define the transition execution sequence based on a transition to keep the semantics consistent with entry behaviours.

Basic Interleave Execution Step. If multiple compound transitions in orthogonal regions are fired by the same event, it is unclear in what granularity should the interleaving execution be conducted: either on transition or on compound transition level. The execution order of the (behaviours associated with the) fired transitions may affect the value of global shared variables. We decide to regard a compound transition as the interleaving execution step, since a compound transition is a semantically complete path.

Order Issues of Entering Orthogonal Composite States. On entering an orthogonal composite state, all possible interleaving orders among its substates to be entered are allowed, as long as the hierarchical order is preserved.

3 Syntax of UML State Machines

In this section, we provide formal syntax definitions for UML state machines features and abstractions of event pools. We define a self-contained model which includes multiple state machines. Table 1 lists the basic notations of types defined in this work.

Our syntax definition preserves the structure specified by [1], which makes it suitable to support refinement as well as future changes of UML state machines.

Table 1. Type notations

Symbol	Type	Symbol	Type	Symbol	Pseudostate type
$\mathcal{K}_\mathcal{S}$	active state configuration	$\mathbb{B}$	boolean	DH_{ps}	deep history
$\widetilde{T}$	compound transition	C	constraints	I_{ps}	initial
$\mathcal{K}$	configurations	S_f	final state	C_{ps}	choice
$\langle \widetilde{T} \rangle$	compound transition list	S	state	Jo_{ps}	join
V	vertex	$Trig$	triggers	Ju_{ps}	junction
$\mathcal{K}_\mathcal{V}$	active vertex configuration	T	transition	T_{ps}	terminate
CR	connection point reference	E	event	En_{ps}	entry point
SM	state machine	R	region	F_{ps}	fork
B	behaviours	PS	pseudostate	SH_{ps}	shallow history
$\langle B \rangle$	behaviour list	$\mathbb{N}$	natural number	Ex_{ps}	exit point

Definition 1 (State). *A* state *is a tuple* $s = (\widehat{r},\ \widehat{t_{def}},\ \alpha_{en},\ \alpha_{ex},\ \alpha_{do},\ \widehat{en},\ \widehat{ex},\ \widehat{cr},\ sm,\ \widehat{t})$ *where:*

- $\widehat{r} \subset R$ *is the set of regions directly contained in this state,*
- $\widehat{t_{def}} \subset Trig$, $\alpha_{en} \in B$, $\alpha_{ex} \in B$ *and* $\alpha_{do} \in B$ *are the set of deferred events, the entry, exit and do behaviours defined in the state, respectively.*
- $\widehat{en} \subset En_{ps}$ *and* $\widehat{ex} \subset Ex_{ps}$ *are the set of entry point and exit point pseudostates associated with the state.*
- $\widehat{cr} \subset CR$ *is the set of connection point references belonging to the state.* $sm \in SM$ *is the state machine referenced by this state; the two fields are used only when the state is a submachine state.*
- $\widehat{t} \subset T$ *is the set of internal transitions defined in the state.*

There are four kinds of states, viz., simple state (S_s), composite state (S_c), orthogonal composite state (S_o) and submachine state (S_m). In Fig. 1, the submachine state *Departure* is denoted as $(\varnothing, \varnothing, \epsilon, \epsilon, \epsilon, \varnothing, \varnothing, \{\textit{EntryP1, ExitP1}\}, \textit{DepartureSM}, \varnothing)$, where ϵ and $\varnothing$ denote the empty element and the empty set, respectively.

Definition 2 (Pseudostate). *A* pseudostate *is a tuple* $ps = (\iota, \widehat{h})$, *where* $\iota \in R \cup SM$ *is the region or state machine in which the pseudostate is defined, and* $\widehat{h} \in S$ *is an optional field which is used to record the last active set of states. This latter field is only used when the pseudostate is a shallow history or deep history pseudostate.*

The last column of Table 1 shows the notations of the ten kinds of pseudostates PS.

Definition 3 (Final state). *A* final state *is a special kind of state, which is defined as a tuple* $s_f = (\iota)$ *where* $\iota \in S_o \cup S_c \cup SM$ *is the composite state or state machine which is the direct ancestor of the container of the final state.*

Definition 4 (Connection Point Reference). *A* Connection Point Reference *is defined as a tuple* $(\widehat{en},\ \widehat{ex},\ s)$ *where* $\widehat{en} \subset En_{ps}$ *and* $\widehat{ex} \subset Ex_{ps}$ *are the entry point and exit point pseudostates corresponding to this connection point reference, and* s *is the submachine state in which the connection point reference is defined.*

For example, in Fig. 1, *EntryP1* is defined as $(\{\textit{EntryPoint1}\}, \varnothing, \textit{DepartureSM})$.
Vertex $V \triangleq S \cup S_f \cup PS \cup CR$ is an abstraction of all nodes.

Definition 5 (Transition). *A* transition *is a tuple* $t = (sv,\ tv,\ \widehat{tg},\ g,\ \alpha\ ,\ \iota,\ \widehat{tc})$ *where:*

- $sv \in V$*,* $tv \in V$ *are the source and target vertex of the transition, respectively.*
- $\widehat{tg} \subset \mathit{Trig}$*,* $g \in C$*,* $\alpha \in B$ *and* $\iota \in R$ *are the set of triggers, the guard, the associated behaviour and the container of the transition, respectively.*
- $\widehat{tc}$ *is a set of tuples of the form* $segt = (ss, \alpha_{st}, \iota_{st})$*. It represents the special situation that a join or fork pseudostate*[2] *connects multiple transitions to form a compound transition. Each tuple represents a segment transition which ends in the join (resp. emanates from the fork) pseudostate.* $ss \in S$ *is the non-fork (resp. non-join) end of the segment transition,* $\alpha_{st} \in B$ *is the behaviour associated with the segment transition.* $\iota_{st} \in R$ *is the container of the segment transition.*

We define the following functions on transitions for clarity sake. Functions $\mathit{isFork}(t)$ and $\mathit{isJoin}(t)$ decide whether transition t is a fork transition and join transition, respectively. For example, in Fig. 1, the join transition *t10* is $(\{\mathit{Join1}\},\ \{\mathit{ExitPoint1}\},\ \varnothing,\ \epsilon,\ \epsilon,\ \mathit{RD},\ \{(\mathit{SyncExit}, \epsilon, \mathit{RD}),\ (\mathit{SyncCruise}, \epsilon, \mathit{RD})\})$. We use $t.\widetilde{\alpha}$ to represent all possible action execution sequences of t. Formal definition of $t.\widetilde{\alpha}$ is in [17].

Definition 6 (Region). *A* region *is defined as a tuple* $r \triangleq (\widehat{v}, \widehat{t})$*, where* $\widehat{v} \subset (S \cup PS \cup S_f)$*,* $\widehat{t} \subset T$ *are the set of vertices and transitions directly owned by the region.*

Definition 7 (State Machine). *A* state machine *is defined as* $sm \triangleq (\widehat{r},\ \widehat{cp})$*, where* $\widehat{r} \subset R$*,* $\widehat{cp} \subset En_{ps} \cup Ex_{ps}$ *are the set of (directly owned) regions and the set of entry/exit point pseudostates defined for this state machine.*

For example in Fig. 1, state machine *DepartureSM* is $(\{\mathit{RD}\}, \{\mathit{EntryPoint1}, \mathit{ExitPoint1}\})$.

Definition 8 (Compound Transition). *A* compound transition *is a "semantically complete" path composed of one or multiple transitions connected by pseudostates. The set of compound transition* $\widetilde{T} = \{\tilde{t} \mid \tilde{t} \in ST \wedge \tilde{t}.\widehat{sv} \in S \wedge \tilde{t}.\widehat{tv} \in S\}$ *where* $st \in ST \equiv (\mathit{len}(st) = 1 \wedge \mathit{seg}(st, 0) \in T) \vee \exists\, st_i, st_j \in ST : \mathit{last}(st_i) = \mathit{first}(st_j) \wedge st = st_i \frown st_j$.

The operator $\frown$ denotes the operation of connecting transitions in order. Notation $\mathit{len}(\tilde{t})$ denotes the total number of segment transitions the compound transition is composed of. $\mathit{seg}(\tilde{t}, i)$ denotes the ith segment specified by the natural number index i of a given compound transition. We use $\mathit{first}(\tilde{t})$ and $\mathit{last}(\tilde{t})$ to denote the first and last segment of $\tilde{t}$. We define $\tilde{t}.\widehat{sv} = \mathit{first}(\tilde{t}).\widehat{sv}$, $\tilde{t}.\widehat{tv} = \mathit{last}(\tilde{t}).\widehat{tv}$ for convenience sake.

Compositional Operators. The operator "; " represents a sequential composition. Interleave operation ($|||$) represents a non-determinism in the execution orders. Interleave with synchronous communications ($|||^C$) is a special case of interleaving: it requires the state machines to synchronize on the specified event in C. Interruption (∇) is used to represent interruption of a do activity by some event occurrence. Parallel composition ($||$) represents a real concurrency, i.e., execute at the same time.

Definition 9 (System). *A* system *is a set of state machines executing in interleaving (with synchronous communications).* $sys \triangleq |||^C_{i \in [1,n]} Sm_i$ *where* $Sm \triangleq (sm, P, GV)$*. In* Sm*,* sm *denotes the state machine,* P *the event pool associated with* sm*, and* GV *the shared variables of* sm*. And* n *is the number of state machines within the system* sys*.*

[2] We treat exit (resp. entry) point pseudostate the same way with join (resp. fork) pseudostate.

For example, the *RailCar* system in Fig. 1 is defined by $|||^C(\mathit{Car}, \mathit{Handler})$, where $C = \{\mathit{departReq}, \mathit{departAck}, \mathit{arriveReq}, \mathit{arriveAck}\}$.[3]

Event Pool Abstraction. Change events, signal events, and deferred events are processed differently in UML state machines. We provide for this purpose 3 separate event pools, viz., completion event pool (CP), deferred event pool (DP), and normal event pool (NP). $P \triangleq (CP, DP, NP)$ represents the event pool, and we define two basic operations on P. $\mathit{Merge}(e, EP)$ merges an event e into the corresponding event pool represented by EP, and $\mathit{Disp}(P)$ dispatches an event from P. Since function Merge (formally defined in [17]) is straightforward, we focus here on Function Disp.

Definition 10. *The following function formally defines the event dispatch mechanism.*

$$\mathit{Disp}(P, ks) \triangleq \begin{cases} CP \backslash \{e\};\ \mathit{CheckDP}(P, ks) & \textit{if } CP \neq \varnothing \wedge \mathit{HighestPriority}(e, CP) \\ DP \backslash \{e\};\ \mathit{CheckDP}(P, ks) & \textit{if } CP = \varnothing \wedge DP \neq \varnothing \wedge !\mathit{isDeferred}(e, ks) \\ NP \backslash \{e\};\ \mathit{CheckDP}(P, ks) & \textit{if } CP = \varnothing \wedge \mathit{allDefer}(DP, ks) \wedge NP \neq \varnothing \\ \epsilon & \textit{otherwise} \end{cases}$$

$$\mathit{CheckDP}(P, ks) \triangleq DP \backslash E;\ NP \cup E,\ \textit{where } E \triangleq \{e \mid e \in DP \wedge !\mathit{isDeferred}(e, ks)\}.$$

The function guarantees that the precedence order $CP \prec DP \prec NP$ is preserved ($\prec$ denotes the preceding partial order). But the order within each event pool is not specified. The macro $\mathit{HighestPriority}(e, CP)$ denotes that event e has the highest priority in CP, which preserves the priority order of a nested state over its ancestor states. In the deferred event pool, only events that are not deferred in the current active state configuration ($!\mathit{isDeferred}(e, ks)$) can be dispatched. The macro $\mathit{allDefer}(DP, ks) \Leftrightarrow \forall\, e \in DP, \mathit{isDeferred}(e, ks)$ guarantees the priority of deferred events over normal events. When an event is dispatched, we check all the deferred events defined in the states of the current active state configuration, and remove those events that are not deferred any more from DP to NP; this is accomplished by $\mathit{CheckDP}$.

4 A Formal Semantics for UML State Machines

This section devotes to a self-contained formal semantics for all UML state machines features. We have adopted the semantic model of Labelled Transition Systems (LTS). The dynamic semantics of a state machine is captured by the execution of RTC steps, which have two kinds of effects, viz., changing active states and executing behaviours. We formally define the two kinds of effects separately. Then the semantics of the RTC step is defined formally. At last, we define the semantics of the system.

4.1 Active State Configuration Changes

An active state configuration $\mathcal{K}_{\mathcal{S}}$ is a set of states which are active at the same time. It describes a stable state status when the previous RTC step finishes. We use Active

[3] We treat the state machine (*DepartureSM*) that is referenced by a submachine state (*Departure*) the same way as a composite state.

Vertex Configuration $\mathcal{K}_{\mathcal{V}}$ (a set of vertices that are active at the same time) to represent the snapshot of a state machine during an RTC execution. For example, in Fig. 1, $\{$*Operating, Choice2*$\}$ is an active vertex configuration. $\mathcal{K}_{\mathcal{S}}$ and $\mathcal{K}_{\mathcal{V}}$ are defined in [17].

Next Active State Configuration. $\mathit{NextK} : \mathcal{K}_{\mathcal{S}} \times \langle \widetilde{T} \rangle \to \mathcal{K}_{\mathcal{S}}$ computes the next active state configuration after executing the compound transition list indicated by $\langle \widetilde{T} \rangle$. Formally: $\mathit{NextK}(ks, (\tilde{t}_1;\ \ldots;\ \tilde{t}_n)) \triangleq \mathit{NxK}(ks_n, \tilde{t}_n)$, where $\forall\, i \in [2, n], ks_i = \mathit{NxK}(ks_{i-1}, \tilde{t}_{i-1}) \wedge ks_1 = ks$. Function $\mathit{NxK} : \mathcal{K}_{\mathcal{S}} \times \widetilde{T} \to \mathcal{K}_{\mathcal{S}}$ computes the next active state configuration after executing a compound transition indicated by $\widetilde{T}$. Formally: $\mathit{NxK}(ks, \tilde{t}) \triangleq \mathit{NxPK}(kv_n, \mathit{seg}(\tilde{t}, n))$, where $n = \mathit{len}(\tilde{t})$, $kv_1 = ks$, and $\forall\, i \in [2, n], kv_i = \mathit{NxPK}(kv_{i-1}, \mathit{seg}(\tilde{t}, i-1))$. Function $\mathit{NxPK} : \mathcal{K}_{\mathcal{V}} \times T \to \mathcal{K}_{\mathcal{V}}$ computes the next active vertex configuration after executing a transition. Formally: $\mathit{NxPK}(kv, t) \triangleq kv \backslash \mathit{Leave}(kv, t) \cup \mathit{Enter}(t)$. Functions *Leave* and *Enter* represent the set of states left and entered after executing a transition and are defined in [17].

4.2 Behaviour Execution

Another effect of executing an RTC step is to cause behaviours to be executed. We define the following functions to collect the behaviour execution sequence.

Exit Behaviour. $\mathit{ExitBehaviour} : \mathcal{K}_{\mathcal{V}} \times T \to \langle B \rangle$ collects the ordered exit behaviours of states that a given transition leaves in the current vertex configuration. Formally:

$$\mathit{ExitBehaviour}(kv, t) = \mathit{ExV}(kv, \mathit{MainSource}(t), t)$$

$$\mathit{ExR}(kv, r, t) \triangleq \begin{cases} \mathit{SH}(h, v);\ \mathit{ExV}(kv, v, t) & \text{if } r \in R \wedge \exists\, v \in r.\widehat{v} : v \in kv \wedge \\ & v \in S \wedge \exists\, h \in \mathit{SH}_{ps} : h \in r.\widehat{v} \\ \mathit{DH}(h, v);\ \mathit{ExV}(kv, v, t) & \text{if } r \in R \wedge \exists\, v \in r.\widehat{v} : v \in kv \wedge v \in S \\ & \wedge\, \exists\, h \in \mathit{DH}_{ps} : \mathit{isAncestor}(h.\iota, r) \\ & \wedge\, \mathit{isAncestor}(t.\iota, h.\iota) \\ \mathit{ExV}(kv, v, t) & \text{if } r \in R \wedge \exists\, v \in r.\widehat{v} : v \in kv \\ & \wedge\, \forall\, s' \in r.\widehat{v}, s' \notin \mathit{SH}_{ps} \\ & \wedge\, \nexists h \in \mathit{DH}_{ps} : \mathit{isAncestor}(h.\iota, r) \\ & \wedge\, \mathit{isAncestor}(t.\iota, h.\iota) \end{cases}$$

$$\mathit{ExV}(kv, v, t) \triangleq \begin{cases} |||^{C}_{r \in v.\widehat{r}} \mathit{ExR}(kv, r, t);\ \mathit{exit}(v) & \text{if } v \in S_o \vee (v \in S_m \wedge v.\widehat{r} \neq \varnothing) \\ \mathit{ExR}(kv, r, t);\ \mathit{exit}(v) & \text{if } v \in S_c \vee (v \in S_m \wedge v.\widehat{r} \neq \varnothing) \\ \mathit{exit}(v) & \text{if } v \in S_s \\ \mathit{ExV}(kv, cr, t) & \text{if } v \in \mathit{Ex}_{ps} \wedge \\ & \exists\, cr \in \mathit{CR} : v \in cr.\widehat{ex} \\ \mathit{ExV}(kv, v.s, t) & \text{if } v \in \mathit{CR} \\ \mathit{Agn}(v.\widehat{r}, v.sm.\widehat{r});\ \mathit{ExV}(kv, v, t) & \text{if } v \in S_m \wedge v.\widehat{r} = \varnothing \\ \epsilon & \text{otherwise} \end{cases}$$

The exit behaviours of executing a transition are collected recursively starting from the innermost state. We define functions *ExV* and *ExR* to recursively collect exit behaviours. All the regions of a composite state should be exited before it. If the region contains a (shallow/deep) history pseudostate, the content of the history pseudostate should be set properly (by functions *SH* and *DH* respectively) before exiting the region. Exiting simple states means terminating the do behaviour (if any) and executing the exit behaviour, as defined by $exit(v) = v.\alpha_{do} \nabla v.\alpha_{ex}$. If an exit point pseudostate is encountered, the associated connection point reference is exited, which means the state defining the connection point reference is exited. Exiting a submachine state means exiting all the regions in the state machine it refers to. Function $Agn(v.\widehat{r}, v.sm.\widehat{r})$ assigns the set of regions of a state machine to the the of regions of a submachine state.

Entry Behaviour. $EntryBehaviour : T \rightarrow \langle B \rangle$ collects the ordered entry behaviours of the states a given transition enters. Formally:
$EntryBehaviour(t) = EnV(MainTarget(t), Enter(t))$
$EnR(r, \widehat{V}) \triangleq EnV(s', \widehat{V})$ where $r \in R \wedge s' \in r.\widehat{v} \wedge s' \in \widehat{V}$

$$EnV(v, \widehat{V}) \triangleq \begin{cases} v.\alpha_{en};\ (|||^{C}_{r \in v.\widehat{r}} EnR(r, \widehat{V}) \parallel v.\alpha_{do}) & \text{if } v \in S_o \vee (v \in S_m \wedge v.\widehat{r} \neq \varnothing) \\ v.\alpha_{en};\ (EnR(r, \widehat{V}) \parallel v.\alpha_{do}) & \text{if } v \in S_c \vee (v \in S_m \wedge v.\widehat{r} \neq \varnothing) \\ v.\alpha_{en};\ v.\alpha_{do} & \text{if } v \in S_s \\ GenEvent(v.\iota) & \text{if } v \in S_f \wedge \forall\, r \in v.\iota.\widehat{r}, \\ & \exists\, s' \in r.\widehat{v} : s' \in kv \Rightarrow s' \in S_f \\ Agn(v.\widehat{r}, v.sm.\widehat{r});\ EnV(v, \widehat{V}) & \text{if } v \in S_m \wedge v.\widehat{r} = \varnothing \\ EnV(v.s, \widehat{V}) & \text{if } v \in CR \\ EnV(cr, \widehat{V}) & \text{if } v \in En_{ps} \wedge \exists\, cr \in CR : v \in cr.\widehat{en} \\ \epsilon & \text{otherwise} \end{cases}$$

Entry behaviours are collected in a similar manner to exit behaviours, except that the collect starts from the outermost state. We define functions EnV and EnR to recursively collect the entry behaviours of all the vertices in $\widehat{V}$ in order. States entered by firing transition t are computed by function $Enter(t)$. Starting from the main target state of a transition, all regions of a composite state are entered in interleaving. Entering each state means executing its entry behaviour followed by its do activities ($s.\alpha_{en};\ s.\alpha_{do}$). Do activities of a composite state should be executed in parallel ($\parallel$) with all the behaviours of its containing states. Function $GenEvent(s)$ generates a completion event for state $s.\iota$ and merges the generated event in the completion event pool (CP).

Collect Actions. $CollectAct : \mathcal{K}_{\mathcal{S}} \times \widetilde{T} \rightarrow \langle B \rangle$ collects the ordered sequence of behaviours associated with the execution of the given compound transition. Formally:
$CollectAct(ks, \tilde{t}) \triangleq Act(kv_1, seg(\tilde{t}, 1));\ \ldots;\ Act(kv_i, seg(\tilde{t}, i));\ \ldots;\ Act(kv_n, seg(\tilde{t}, n))$, $andAct(kv, t) \triangleq ExitBehaviour(kv, t);\ t.\widetilde{\alpha};\ EntryBehaviour(t)$ where $n = len(\tilde{t})$, $kv_1 = ks$ and $kv_i = NxPK(kv_{i-1}, seg(\tilde{t}, i-1))$ for $i \in [2, n]$.

4.3 The Run to Completion Semantics

The effects of an RTC step execution include both active state changes and behaviour executions which may cause the event pool and global shared variables to be updated. We use the term *configuration* to capture the stable status of a state machine.

Definition 11. *A* configuration *is a tuple $k = (ks, P, GV)$ where ks is the active state configuration, P is the event pool and GV is the set of valuation of global variables.*

For example, $(\{Idle\}, (\varnothing, \varnothing, \{setDest\}), \{stopNum = 0, mode = false\})$ is a configuration. The execution of an RTC step can be depicted as moving from one configuration to the next configuration. We provide the following rules to formalize an RTC step. We use the *RailCar* system in Fig. 1 to illustrate the following RTC step rules.

Wandering Rule. This rule captures the case where a dispatched event e is neither consumed nor delayed. As a result, it is discarded.

$$\frac{e = \mathit{Disp}(P), P' = P\backslash\{e\}, \forall\, s \in \mathit{ks}, e \notin s.\widehat{t_{\mathit{def}}}, \mathit{Enable}((\mathit{ks}, P', GV), e) = \varnothing}{(\mathit{ks}, P, GV) \xrightarrow{e} (\mathit{ks}, P', GV)}$$

Event e is dispatched from event pool ($\mathit{Disp}(P)$), but no transition is triggered by e (i.e., $\mathit{Enable}((\mathit{ks}, P', GV), e) = \varnothing$), and no deferred event in the current configuration matches the event e (i.e., $\forall\, s \in \mathit{ks}, e \notin s.\widehat{t_{\mathit{def}}}$).

Deferral Rule 1. This rule captures the case where a dispatched event is deferred by some states in the current active state configuration, but does not trigger any transitions.

$$\frac{\begin{array}{l} e = \mathit{Disp}(P), P' = P\backslash\{e\}, \exists\, s \in \mathit{ks} : e \in s.\widehat{t_{\mathit{def}}}, \mathit{Enable}((\mathit{ks}, P', GV, e) = \varnothing, \\ P'' = \mathit{Merge}(e, P'.\mathit{DP}) \end{array}}{(\mathit{ks}, P, GV) \xrightarrow{e} (\mathit{ks}, P'', GV)}$$

Since event e is deferred, it should be merged back to the deferred event pool (i.e., $\mathit{Merge}(e, P'.\mathit{DP})$). So after the RTC execution, only the event pool is changed to P''.

Deferral Rule 2. This rule captures the case where the dispatched event e triggers some transitions and it is also deferred by some states in the current active state configuration. But there exists at least one state, which defines the deferred event, that has higher priority than the source states of the enabled transitions.

$$\frac{\begin{array}{l} e = \mathit{Disp}(P), P' = P\backslash\{e\}, \exists\, s \in \mathit{ks} : e \in s.\widehat{t_{\mathit{def}}}, \widehat{T} = \mathit{Enable}((\mathit{ks}, P', GV, e), \widehat{T} \neq \varnothing, \\ \forall\, \tilde{t} \in \widehat{T} \Rightarrow \mathit{deferralConflict}(\tilde{t}, (\mathit{ks}, P', GV), e), P'' = \mathit{Merge}(e, P'.\mathit{DP}) \end{array}}{(\mathit{ks}, P, GV) \xrightarrow{e} (\mathit{ks}, P'', GV)}$$

$\widehat{T}$ is the set of transitions enabled by the dispatched event e. Event e is also deferred by some states in the current active state configuration and the event deferral has higher priority over transition firing ($\forall\, \tilde{t} \in \widehat{T} \Rightarrow \mathit{deferralConflict}(\tilde{t}, (\mathit{ks}, P', GV), e)$)). As a consequence, only the event pool of the state machine changes. For example, $(\{\mathit{Operating, WaitArriveOK, Watch, WaitEnter}\}, (\varnothing, \varnothing, \{\mathit{opend}\}), \mathit{Env}1) \xrightarrow{\mathit{opend}}$ $(\{\mathit{Operating, WaitArriveOK, Watch, WaitEnter}\}, (\varnothing, \{\mathit{opend}\}, \varnothing), \mathit{Env}1)$ illustrates the application of this rule, where $\mathit{Env}1$ denotes $\{\mathit{stopNum} = 1, \mathit{mode} = \mathit{false}\}$.

To increase the rules readability, we use the following notations. $\mathcal{A}(\tilde{t}_1, \ldots, \tilde{t}_n) = \mathit{CollectAct}(\tilde{t}_1);\ ,\ldots,;\ \mathit{CollectAct}(\tilde{t}_n)$ denotes the behaviours collection along transitions $\tilde{t}_1, \ldots, \tilde{t}_n$. $\mathit{Merge}(\mathcal{A}(\langle\tilde{t}\rangle), P)$ merges all events generated by actions in $\mathcal{A}(\langle\tilde{t}\rangle)$ into event pool P. Function $\mathit{UpdateV}(\mathcal{A}(\langle\tilde{t}\rangle), GV)$ updates global variables GV by actions in $\mathcal{A}(\langle\tilde{t}\rangle)$.

Progress Rule. This rule captures the case where a set of compound transitions are triggered by a dispatched event e. There is no event deferred, or the fired transitions have higher priority over event deferral.

$$\frac{\begin{array}{l} e = \mathit{Disp}(P), P' = P \backslash \{e\}, \widehat{T} \in \mathit{Firable}((\mathit{ks}, P', GV), e), \mid \widehat{T} \mid = n, \\ \langle \tilde{t} \rangle \in \mathit{Permutation}(\widehat{T}), P'' = \mathit{MergeA}(\mathcal{A}(\langle \tilde{t} \rangle), P'),\ V' = \mathit{UpdateV}(\mathcal{A}(\langle \tilde{t} \rangle), GV) \end{array}}{(\mathit{ks}, P, GV) \xrightarrow{e} (\mathit{NextK}(\mathit{ks}, \langle \tilde{t} \rangle), P'', GV')}$$

Function $\mathit{Firable}((\mathit{ks}, P', GV), e)$ (defined in [17]) returns a set of maximal non-conflicting subset of enabled transitions. The firable set of transitions[4] will be executed in an order specified by $\langle \tilde{t} \rangle$. Function $\mathit{Permutation}$ (defined in [17]) computes all possible total orders on the set of compound transitions $\widehat{T}$. Behaviours are collected along the transition execution sequence following the permutation order (indicated by $\mathcal{A}(\langle \tilde{t} \rangle)$). Active state configuration is changed as computed by function $\mathit{NextK}(\mathit{ks}, \langle \tilde{t} \rangle)$.

ProgressC Rule. This rule captures the case where choice pseudostates are encountered during an RTC execution. Different from the RTC Progress rule, dynamic evaluation would be conducted at the point where a choice pseudostate is reached.

$$\frac{\begin{array}{l} e = \mathit{Disp}(P), P' = P \backslash \{e\}, \widehat{T} \in \mathit{Firable}((\mathit{ks}, P', GV), e), \mid \widehat{T} \mid = n, \\ \tilde{t}_i^1 \in \widehat{T}, \tilde{t}_i^1.\mathit{tv} \in C_{ps}, \langle \tilde{t} \rangle = (\tilde{t}_1, \dots \tilde{t}_i^1, \dots, \tilde{t}_n) \in \mathit{Permutation}(\widehat{T}), \\ GV' = \mathit{UpdateV}(\mathcal{A}(\tilde{t}_1, \dots, \tilde{t}_i^1)), GV), P'' = \mathit{MergeA}(\mathcal{A}(\tilde{t}_1, \dots, \tilde{t}_i^1)), P'), \\ \tilde{t}_i^2 \in \mathit{Firable}((\{\mathit{last}(\tilde{t}_i^1).\mathit{tv}\}, P'', GV'), e), P''' = \mathit{MergeA}(\mathcal{A}(\tilde{t}_i^2 \dots, \tilde{t}_n), P''), \\ GV'' = \mathit{UpdateV}(\mathcal{A}(\tilde{t}_i^2 \dots, \tilde{t}_n), GV') \end{array}}{(\mathit{ks}, P, GV) \xrightarrow{e} (\mathit{NextK}(\mathit{ks}, \langle \tilde{t} \rangle), P''', GV'')}$$

Compound transition t_i is split by a choice pseudostate into t_i^1 and t_i^2. The second half of t_i is evaluated based on environment GV' . In Fig. 1, $(\{\mathit{Operating}, \mathit{WaitArriveOK}, \mathit{Watch}, \mathit{WaitDepart}\}, (\varnothing, \varnothing, \{\mathit{opend}\}), \mathit{Env}1) \overset{\mathit{opend}}{\dashrightarrow} (\{\mathit{Operating}, \mathit{Choice2}\}, (\varnothing, \varnothing, \varnothing), \mathit{Env}0) \dashrightarrow (\{\mathit{Idle}\}, (\varnothing, \varnothing, \varnothing), \mathit{Env}0)$[5] illustrates the application of this rule.

4.4 System Semantics

A UML state machine models the dynamic behaviour of one object within a system. But state machines representing different components of a system may interact with each other. In order to verify the correctness of the overall system behaviours, we need to capture the message passing sequences between state machines in the system.

Definition 12 (Semantics of a system). *The* semantics of a system *is defined as a Labelled Transition System (LTS)* $\mathcal{L} \triangleq (\mathbb{S}, \mathcal{S}_{init}, \rightsquigarrow)$. *In this expression,* $\mathbb{S}$ *is the set of states of* $\mathcal{L}$. *Each LTS state is a tuple* $(k_1, \dots, k_n)$ *where* k_i *is the configuration of the state machine* Sm_i *within the system.* $\mathcal{S}_{init}$ *is the initial state of* $\mathcal{L}$. *And* $\rightsquigarrow \subseteq \mathbb{S} \times \mathbb{S}$ *is the transition relation of* $\mathcal{L}$, *defined below.*

[4] We assume the UML state machines obey well-formedness rules. If more than one non-conflicting sets of transitions are fiable, the choice of which set to execute is non-deterministic.

[5] We use $\mathit{Env}0$ to represent the set $\{\mathit{stopNum} = 0, \mathit{mode} = \mathit{false}\}$. The dashed arrow $\dashrightarrow$ represents an instant stop in a choice pseudostate.

Table 2. Evaluation results

Model	Property	Result	USM^{2}C				HUGO				
			Time(s)	State	Transition	Mem (KiB)	TTime(s)	ETime(s)	State	Transition	Mem (KiB)
RailCar	Prop1	not valid	0.013	30	34	$43,342$	-	-	-	-	-
RailCarO	Prop1	valid	0.011	44	54	$43,058$	-	-	-	-	-
BankATM	Prop2	valid	0.009	25	28	917.5	0.231	0.050	578	$1,133$	$98,528$
DP2	deadlock	not valid	0.005	39	65	$2,318$	0.196	0.111	$12,766$	$42,081$	$98,918$
TollGate	Prop3	valid	0.110	36	50	$43,345$	0.197	0.505	$61,451$	$256,807$	$100,578$

$$\frac{|||^{C}_{i\in[1,n]} Sm_i, k_j \to k_j'}{(k_1,\ldots,k_j,\ldots,k_n) \rightsquigarrow (k_1,\ldots,k_j',\ldots,k_n)}\ [\ LTS1\ ]$$

$$\frac{|||^{C}_{i\in[1,n]} Sm_i, k_j \to k_j', e = SendSignal(j,l), Merge(e, EP_l)}{(k_1,\ldots,k_l,\ldots,k_j,\ldots,k_n,) \rightsquigarrow (k_1,,\ldots,k_l',\ldots,k_j',\ldots,k_n)}\ [\ LTS2\ ]$$

$$\frac{|||^{C}_{i\in[1,n]} Sm_i, k_j \to k_j', e = Call(j,l), e \in C, k_l \xrightarrow{e} k_l'}{(k_1,\ldots,k_l,\ldots,k_j,\ldots,k_n) \rightsquigarrow (k_1,\ldots,k_l',\ldots,k_j',\ldots,k_n)}\ [\ LTS3\ ]$$

All the state machines in the system are executed non-deterministically. Rule LTS1 captures the normal situation that a single state machine is executed without communicating with other state machines. The notation with prime, i.e., k_j', represents the new configuration after executing an RTC step. Rule LTS2 defines asynchronous communication, i.e., the executing state machine (Sm_j) sends an asynchronous message ($e = SendSignal(j,l)$) to another state machine (Sm_l). The state machine receiving the message merges the message into its own event pool. Rule LTS3 defines synchronous communication. In this case, the callee state machine (Sm_l) is triggered by the call event ($e = Call(j,l)$, $e \in C$). The caller state machine (Sm_j) cannot finish its RTC step until the callee has finished execution. For example in Fig. 1, if state machine *Car* and *Handler* are in configuration $(\{$*Operating, Crusing*$\}, (\varnothing, \varnothing, \{$*alert100*$\}, Env1)$, $(\{$*WaitDepart*$\}, (\varnothing, \varnothing, \varnothing), \varnothing)$ separately and event *alert100* is dispatched and fires transition t_{12}. The behaviour associated with t_{12} invokes a call event (that is *arriveReq* $=$ *Call*(*Car*, *Handler*)) in *Handler* state machine. The *Handler* state machine consumes the call event and execute an RTC step. After applying rule LTS3, the system is $((\{$*Operating, WaitArriveOK, Watch, WaitEnter*$\}, (\varnothing, \varnothing, \varnothing), Env1), (\{$*WaitPlatform*$\}, (\varnothing, \varnothing, \varnothing), \varnothing))$.

5 Implementation and Evaluation

We have implemented the formal semantics in a self-contained tool USM^{2}C [2]. It supports model checking of deadlock, LTL properties, and step-wise simulation. . We compared USM^{2}C with HUGO [13] on 5 examples used in literature, viz., *RailCarO* [10],

Table 3. Scalability evaluation result

N	Time (s)	States	Transitions	Memory (KiB)	N	Time (s)	States	Transitions	Memory (KiB)
2	0.005	39	65	2,318	3	0.039	237	589	10,145
4	0.34	1,519	5,079	21,059	5	3.11	9,634	40,366	41,651
6	27.87	63,069	324,275	90,023	7	232.64	398,101	2,385,361	2,852,672

RailCar in Fig. 1 (modifies *RailCarO* to manually introduce bugs[6]), *BankATM* [13], dining philosopher ($n = 2$) and *TollGate* [15]. HUGO is a tool translating UML state machines into Promela models and using Spin to perform model checking.

Results are in Table 2, where Prop1=$\Box(alert100 \rightarrow \Diamond arriveAck)$, Prop2=$\Box(retain \rightarrow ((!cardValid \wedge numIncorrect \geq maxNumIncorrect))$, Prop3=$\Box(TurnGreen \rightarrow \Diamond carExit)$. Our tool finds the manually injected bugs in *RailCar* system, which is out of the capability of HUGO. The results also show that our tool is more efficient in execution time and memory consumption compared to HUGO[7]. The main reason is that the Promela code generated by HUGO has many local transitions, which introduce overheads. For example, in the generated *TollGate* promela code, 7 steps are conducted to move from a initial pseudostate to its target state, while in our model only one (implicit) step is taken. The effect is exponential in case of non-determinism.

We conducted another experiment on the dining philosophers problem to evaluate the scalability of USM^{2}C. Table 3 shows the result of checking deadlock free property (with breadth first search). We can see from the result that USM^{2}C can handle large state spaces caused by non-determinism. Reducing further the state space through techniques such partial-order reduction is the subject of our future work.

We believe that communications between objects are error-prone and hard to find manually. The experiment results show that our method can find design errors in the presence of both synchronous and asynchronous communications and is scalable.

6 Related Work

Existing approaches for formalizing UML state machines semantics fall into two major groups, viz., translation-based approaches and direct formalization approaches.

A large number of existing approaches translate UML state machines to an existing formal modelling language, such as Abstract State Machines [11,5,12], Petri nets [6,3], or the modelling language of some model checkers. The verification can be accomplished by relying on verification tools for the translated languages. For example, state machines have been translated to Promela [14], CSP [18], Event-B [21] and CSP# [23]; then Spin, FDR, ProB and PAT model checkers are used to perform the verification, respectively. The translation approaches suffer from the following defects: (1) Due to the semantic gaps, it may be hard to translate some syntactic features of UML state machines, introducing sometimes additional but undesired behaviours. For example

[6] Both examples contain transitions which emanate/enter orthogonal composite states, e.g., the transition from *Cruising* state to *WaitArrivalOK* state, which is not supported by HUGO.

[7] TTime represents the time used to translate UML state machines models into Promela. ETime represents the time used by Spin to do model checking.

in [23], extra events have to be added to each process so as to model exit behaviours of orthogonal composite states. (2) For the verification, translation approaches heavily depend on the tool support of the target formal languages. Furthermore, the additional behaviours introduced during the translation may significantly slow down the verification; and optimizations and reduction techniques (like partial order reduction) may not apply in order to preserve the semantics of the original model. (3) Lastly, when a counterexample is found by the verification tool, it is hard to map it to the original state machine execution, especially when state space reduction techniques are used.

Works directly provide operational semantics for UML state machines are more related to our approach. [22] provides an operational semantics for a subset of UML state machines. The approach uses terms to represent states and transitions are nested into Or-terms, which makes it hard to extend to support the other features. Fecher [7] defines a formal semantics for a subset of UML state machines features. The remaining features are informally transformed to the formalized features. The informal transformation procedure as well as the extra costs it introduces might make it infeasible for tool developing. The work in [19] considers non-determinism in orthogonal composite states. But it supports only a subset of features and neither event pool mechanisms nor RTC steps are discussed. In all those works [22,7,19], behaviours associated with states and transitions are explicitly represented with mapping functions. As a consequence, future changes to the state machines may cause modifications of multiple structures in their syntax definition and the consistencies between those structures need to be properly maintained. Conversely, our semantics preserves the syntax structure specified by the specification and should extend better to future changes and refinements of state machines. For example, if a simple state is refined into a composite state, only the definition of that simple state needs to be changed in our approach, whereas all the mappings related to that simple state need to be changed in their work.

A number of prototype tools were developed to support the verification of UML state machines in the literature. vUML [16] and HUGO [13] are tools that translate UML state machines to PROMELA and use Spin for the verification. TABU [4] and the tool proposed in [20] translate UML state machines to the input language of SMV. JACK [9] is an integrated environment containing an AMC component, which is able to conduct model checking. UML-B [21] is developed to support translation from UML state machines into Event-B model and ProB is invoked to conduct model checking. Among all the tools discussed here, only HUGO and UML-B are currently available. HUGO has compatibility problems with newer versions of Spin (Spin5.x, Spin6.x), thus manual efforts and knowledge of Spin are required for the verification. UML-B is a UML-like notation, which integrates with B.

7 Discussion and Perspectives

In this paper, we provide a formal semantics for the complete set of UML state machines features. Our semantics considers non-determinism as well as the communication aspects between UML state machines, which bridge the gap of current approaches. We have implemented a self-contained tool, USM^{2}C, for model checking various properties for UML behavioural state machines. The experiments show that our tool is effective in finding bugs with communications between different state machines.

We discuss in the following limitations related to our work. (1) We provide basic assumptions for those unclarities found in UML 2.4.1 state machines specifications based on our understanding, which may introduce thread to the validity of our work. (2) We did not formally define the constraint and action language in this work.

Several other issues linked with UML state machines remain unaddressed. As future work, we aim at considering the real-time aspects and object-oriented issues, such as dynamic invoking and destroying objects.

Acknowledgements. We thank the anonymous reviewers for their insightful comments.

References

1. OMG unified language superstructure specification (formal), Version 2.4.1 (August 06, 2011), `http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/`.
2. USM^2C, a UML state machines model checker (April 05, 2013), `http://www.comp.nus.edu.sg/~lius87`
3. André, É., Choppy, C., Klai, K.: Formalizing non-concurrent UML state machines using colored Petri nets. ACM SIGSOFT Software Engineering Notes 37(4), 1–8 (2012)
4. Beato, M.E., Barrio-Solórzano, M., Cuesta, C.E., Fuente, P.: UML automatic verification tool with formal methods. Elec. N. in Th. Computer Sc. 127(4), 3–16 (2005)
5. Börger, E., Cavarra, A., Riccobene, E.: On formalizing UML state machines using ASMs. Information Software Technology 46(5), 287 (2004)
6. Choppy, C., Klai, K., Zidani, H.: Formal verification of UML state diagrams: a Petri net based approach. ACM SIGSOFT Software Engineering Notes 36(1), 1–8 (2011)
7. Fecher, H., Schönborn, J.: UML 2.0 state machines: Complete formal semantics via core state machine. Formal Methods: Applications and Technology, 244–260 (2007)
8. Fecher, H., Schönborn, J., Kyas, M., de Roever, W.: 29 new unclarities in the semantics of UML 2.0 state machines. Formal Methods and Software Engineering, 52–65 (2005)
9. Gnesi, S., Latella, D., Massink, M.: Model checking UML statechart diagrams using JACK. In: HASE 1999, pp. 46–55 (1999)
10. Harel, D., Gery, E.: Executable object modeling with statecharts. IEEE Computer 30, 31–42 (1997)
11. Jin, Y., Esser, R., Janneck, J.: A method for describing the syntax and semantics of UML statecharts. Software and Systems Modeling 3(2), 150–163 (2004)
12. Jürjens, J.: A UML statecharts semantics with message-passing. In: Proceedings of the 2002 ACM Symposium on Applied Computing, pp. 1009–1013. ACM (2002)
13. Knapp, A., Merz, S.: Model checking and code generation for UML state machines and collaborations. In: Proc. 5th W. Tools System Design & Verif, vol. 11, pp. 59–64 (2002)
14. Knapp, A., Merz, S., Rauh, C.: Model checking - timed UML state machines and collaborations. In: Damm, W., Olderog, E.-R. (eds.) FTRTFT 2002. LNCS, vol. 2469, pp. 395–416. Springer, Heidelberg (2002)
15. Kong, J., Zhang, K., Dong, J., Xu, D.: Specifying behavioral semantics of UML diagrams through graph transformations. Journal of Systems and Software 82(2), 292–306 (2009)
16. Lilius, J., Paltor, I.P.: vUML: A tool for verifying UML models, pp. 255–258 (1999)
17. Liu, S., Liu, Y., André, É., Choppy, C., Sun, J., Wadhwa, B., Dong, J.S.: A formal semantics for complete UML state machines with communications (report). Technical report, National University of Singapore (2013), `http://www.comp.nus.edu.sg/ lius87/uml/techreport/uml_sm_semantics.pdf`

18. Ng, M., Butler, M.: Towards formalizing UML state diagrams in CSP. In: SEFM 2003, p. 138 (2003)
19. Schönborn, J.: Formal semantics of UML 2.0 behavioral state machines. Technical report, Inst. Computer Science and Applied Mathematics, Christian-Albrechts-Univ. of Kiel (2005)
20. Shen, W., Compton, K., Huggins, J.: A toolset for supporting UML static and dynamic model checking. In: COMPSAC 2002, pp. 147–152 (2002)
21. Snook, C., Butler, M.: UML-B: Formal modeling and design aided by UML. ACM Trans. Softw. Eng. Methodol. 15(1), 92–122 (2006)
22. Von Der Beeck, M.: A structured operational semantics for UML-statecharts. Software and Systems Modeling 1(2), 130–141 (2002)
23. Zhang, S., Liu, Y.: An automatic approach to model checking UML state machines. In: 4th Int. Conf. Secure Software Integration & Reliability etc (SSIRI-C), pp. 1–6. IEEE (2010)

From Small-Step Semantics to Big-Step Semantics, Automatically*

Ştefan Ciobâcă

Faculty of Computer Science, University "Alexandru Ioan Cuza"
Iaşi, Romania
stefan.ciobaca@info.uaic.ro

Abstract. Small-step semantics and big-step semantics are two styles for operationally defining the meaning of programming languages. Small-step semantics are given as a relation between program configurations that denotes one computational step; big-step semantics are given as a relation directly associating to each program configuration the corresponding final configuration. Small-step semantics are useful for making precise reasonings about programs, but reasoning in big-step semantics is easier and more intuitive. When both small-step and big-step semantics are needed for the same language, a proof of the fact that the two semantics are equivalent should also be provided in order to trust that they both define the same language. We show that the big-step semantics can be automatically obtained from the small-step semantics when the small-step semantics are given by inference rules satisfying certain assumptions that we identify. The transformation that we propose is very simple and we show that when the identified assumptions are met, it is sound and complete in the sense that the two semantics are equivalent. For a strict subset of the identified assumptions, we show that the resulting big-step semantics is sound but not necessarily complete. We discuss our transformation on a number of examples.

1 Introduction

In order to reason about programs, a formal semantics of the programming language is needed. There exist a number of styles in which the semantics can be given: denotational [1] (which associates to a program a mathematical object taken to be its meaning), axiomatic [2,3] (where the meaning of a program is exactly what can proven about it from the set of axioms) and operational [4,5,6] (which describes how the program executes on an abstract machine). Each semantic style has its own advantages and disadvantages. Sometimes, two or more semantics in different styles are needed for a programming language. In such a case, the equivalence of the different semantics must be proven in order to be sure that they really describe the same language.

In this article, we focus on two (sub)styles of operational semantics: small-step structural operational semantics [5] and big-step structural operational semantics [6] (also called natural semantics). Small-step semantics are given as

* This work was supported by Contract 161/15.06.2010, SMISCSNR 602-12516 (DAK).

a relation ($\to$) between program configurations modelling code and state. The small-step relation is usually defined inductively (based on the structure of the abstract syntax of the programming language – therefore the name of *structural* operational semantics). Configurations in which programs are expected to end are called *final configurations* (e.g., configurations in which there is no more code to execute). The transitive-reflexive closure $\to^*$ of the small-step rewrite relation $\to$ is taken to model the execution of a program. Configurations that cannot take a small step but which are not final configurations are *stuck* (e.g., when the program is about to cause a runtime error such as dividing by zero).

In contrast, big-step (structural operational) semantics describe the meaning of a programming language by an inductively defined predicate $\Downarrow$ which links a (starting) program configuration directly to a *final* configuration. Therefore the big-step semantics of a programming language is in some sense similar to the transitive closure of the small-step semantics.

Small-step semantics are especially useful in modeling systems with a high degree of non-determinism such as concurrent programming languages and in proofs of soundness for type systems, since small-step semantics can distinguish between programs that go wrong (by trying to perform an illegal operation such as adding an integer to a boolean) and programs which do not terminate because they enter an infinite loop. In contrast, a big-step semantics has the disadvantage that it cannot distinguish between a program that does not terminate (such as the program $\mu x.x$ in a lambda calculus extended with the fix-point operator μ) and a program that performs an illegal operation (such as the program 1 2 – the program which tries to apply the natural number 1 (which is not a function and therefore cannot be applied to anything) to the natural number 2 – in a lambda calculus extended with integers). The reason for which the big-step semantics cannot distinguish between these is that in both cases there does not exist a final configuration M such that $\mu x.x \Downarrow M$ or $1\ 2 \Downarrow M$. Furthermore, big-step semantics cannot be used to model non-determinism accurately: consider a language with a C-like *add-and-assign* operator $+=$ and the following statement: $x := (x \mathrel{+=} x+1) + ((x \mathrel{+=} x+2) + (x \mathrel{+=} x+3))$. The *add-and-assign* operator evaluates the right-hand side, adds the result to the variable on the left-hand side and returns the new value of the variable. If the language has a non-deterministic $+$ operator, then the evaluation of the three *add-and-assign* expressions can happen in any order. However, the following (slightly simplified) big-step rules which seem to naturally model the non-determinism of $+$:

$$\frac{}{V_1 + V_2 \Downarrow V}\ V = V_1 +_{Int} V_2 \qquad \frac{M \Downarrow V_1 \quad V_1 + N \Downarrow V}{M + N \Downarrow V} \qquad \frac{N \Downarrow V_2 \quad M + V_2 \Downarrow V}{M + N \Downarrow V}$$

will fail to capture all behaviors of the statement, since the side-effects of $x \mathrel{+=} x+1$ will never be taken into account *in the middle of the evaluation* of $((x \mathrel{+=} x+2) + (x \mathrel{+=} x+3))$. This is a known inherent limitation of big-step semantics which prevents it from being used to reason about concurrent systems. Known workarounds [7] that allow a certain degree of non-determinism in big-step semantics require adding additional syntax to the language (required only for the evaluation), which makes the semantics less structural.

Big-step semantics do have however a few notable advantages. First of all, they can be used to produce efficient interpreters and compilers [8] since there is no need to search for the redex to be reduced – instead, the result of a program is directly computed from the results of smaller programs. In contrast, an interpreter based on small-step semantics has to search at each step for a possible redex and then perform the update. Secondly, reasoning about programs and about the correctness of program transformations with a big-step semantics is easier [9,10].

Therefore, because both small-step semantics and big-step semantics each have their own set of advantages, it is desirable to have both types of semantics for a programming language. However, when both the small-step semantics and the big-step semantics are given for a language, the equivalence of the two semantics needs to be proven in order to be sure that the two semantics define the same language. We would like to obtain the advantages of both small-step semantics and big-step semantics, but without having to do this proof (or at least, not having to redo it for every programming languages being defined).

Therefore, we propose and investigate a transformation to *automatically* obtain the big-step semantics of a language from its small-step semantics. This allows in principle to enjoy both the advantages of the small-step semantics and those of the big-step semantics without having to manually maintain the two semantics and their proof of equivalence. Of course, this automation does not come without costs: in order for the transformation to yield an equivalent semantics, a number of assumptions must hold for the small-step semantics.

Our motivation for transforming small-step semantics into big-step semantics comes from our research on the K Semantic Framework [11], which is a framework for defining programming languages based on rewriting logic [12]. The K framework can be seen as a methodological fragment of rewriting logic with rewrite rules that describe small-step semantics and heating and cooling rules which describe under which contexts the rules can apply. We intend to generate standalone compilers for efficient execution and mechanized formal specifications for proof assistants in order to perform machine-assisted reasoning about programs. This transformation could serve as a starting point for both these directions.

In Section 2, we describe the meta-language we use for the small-step and big-step semantics. In Section 3, we formalize our transformation and present all of the assumptions under which it is sound and complete. In Section 4 we present a number of examples and in Section 5 we present related work. Section 6 contains a discussion and directions for further work.

2 Preliminaries

Before formalizing our transformation from small-step semantics to big-step semantics, we need a precise mathematical language (the meta-language) to describe such semantics. As previously discussed, the small-step semantics of a programming language is a binary relation $\rightarrow$ between program configurations. In the following, we model ground program configurations by an arbitrary algebra $\mathcal{A}$ over a signature Σ. Abstract configurations (i.e. configurations with variables, simply "configurations" from here on) are built from the signature Σ and a countably infinite set of variables $\mathcal{X}$ as expected. Substitutions σ are defined as expected and substitution application is written in suffix form. We use the letters M, N, P and their decorated counterparts (M_i, M^i, M_i^j, etc) as *meta-variables* in the meta-language; i.e., they can denote any particular program configuration.

Example 1. To define the untyped lambda calculus, we consider the signature $\Sigma = \{\mathsf{app}, \mathsf{fun}, x_0, \ldots, x_n, \ldots\}$. The constants $x_0, \ldots, x_n, \ldots$ denote the variables of lambda calculus, the binary symbol fun denotes functional abstraction and the binary symbol app denotes application. We follow the usual notations in lambda calculi and we write applications $\mathsf{app}(M, N)$ as juxtapositions MN and functional abstractions $\mathsf{fun}(x_i, M)$ as lambda-abstractions $\lambda x_i.M$. The algebra $\mathcal{A}$ is then the initial algebra of Σ. We assume that $\mathcal{A}$ is sorted such that the first argument of fun is always a constant x_i $(i \in N)$. □

We let $\mathcal{P}$ be a set of predicates. We assume that $\mathcal{P}$ contains the distinguished binary predicates $\rightarrow$ and $\Downarrow$ and the distinguished unary predicate $\downarrow$. The predicates $\rightarrow$ and respectively $\Downarrow$ are used in infix notation and the predicate $\downarrow$ is used in suffix notation. The predicate $\rightarrow$ is reserved for the small-step transition relation, the predicate $\Downarrow$ is used for the big-step relation and $\downarrow$ is used for denoting final configurations.

Example 2. Continuing the previous example, for call-by-value lambda calculus (CBV lambda calculus), the predicate $\downarrow$ (denoting final configurations) is defined to be true only for configurations that are either variables or lambda-abstractions:

$$M{\downarrow} \textit{ iff } \begin{cases} M = x_i & \textit{or} \\ M = \lambda x_i.N \end{cases} \quad \textit{(for some } i \in \mathbb{N},\ N \in \mathcal{A}\textit{)}.$$

In the context of lambda calculi, we also consider a predicate $\mathit{Subst}(M, x, N, P)$ which is true when P is a lambda-term obtained by substituting N for the variable x in M while avoiding name-capture. □

We model the small-step semantics as a set of inference rules R of the form

$$R = \frac{M_1 \rightarrow N_1, \ldots, M_n \rightarrow N_n}{M \rightarrow N}\ Q_1(\tilde{P}_1), \ldots, Q_m(\tilde{P}_m),$$

where $M, N, M_1, \ldots, M_n, N_1, \ldots, N_n$ are configurations, $\tilde{P}_1, \ldots, \tilde{P}_m$ are sequences of configurations and $Q_1, \ldots, Q_m \in \mathcal{P} \setminus \{\rightarrow, \Downarrow\}$ are predicates. The transition relation $\rightarrow$ associated to such a set of inference rules is the smallest relation which is closed by each inference rule, i.e., for each rule R as above and each substitution σ grounding for R, we have that if $M_1\sigma \rightarrow N_1\sigma, \ldots, M_n\sigma \rightarrow N_n\sigma$ and $Q_1(\tilde{P}_1\sigma), \ldots, Q_m(\tilde{P}_m\sigma)$ then $M\sigma \rightarrow N\sigma$.

Example 3. Continuing the previous example, the small-step semantics of call-by-value lambda calculus can be defined by the following set $S = \{R_1, R_2, R_3\}$ of inference rules:

$$(R_1)\ \frac{X \to X'}{XY \to X'Y} \qquad (R_2)\ \frac{Y \to Y'}{XY \to XY'}\ \mathrm{X}{\downarrow} \qquad (R_3)\ \frac{}{(\lambda x.X)Y \to Z}\ Y{\downarrow}, \mathit{Subst}(X, x, Y, Z)$$

Note that in the above rules, $x, X, X', Y, Y', Z \in \mathcal{X}$ are variables; we assume $x \in \mathcal{X}$ is sorted to be instantiated only with lambda-calculus variables $x_i \in \Sigma$. □

As a sanity check for the definition of small-step semantics, it is expected that $M{\downarrow}$ implies $M \not\to N$ for any N (i.e. configurations that are considered final cannot take any step). The reverse implication is not expected, since a configuration such as $x_0 x_1$ (application of the variable x_0 to the variable x_1) is *stuck* and cannot advance even if it not a final configuration. Similarly to small-step semantics, big-step semantics are modeled as a set of inference rules R of the form

$$R = \frac{M_1 \Downarrow N_1, \ldots, M_n \Downarrow N_n}{M \Downarrow N}\ Q_1(\tilde{P}_1), \ldots, Q_m(\tilde{P}_m),$$

where $M, N, M_1, \ldots, M_n, N_1, \ldots, N_n$ are configurations, $\tilde{P}_1, \ldots, \tilde{P}_m$ are sequences of configurations and $Q_1, \ldots, Q_m \in \mathcal{P} \setminus \{\to, \Downarrow\}$ are predicates. The relation $\Downarrow$ associated to such a set of inference rules is the smallest relation which is closed by each inference rule, i.e., for each rule R as above and each substitution σ grounding for R, we have that if $M_1\sigma \Downarrow N_1\sigma, \ldots, M_n\sigma \Downarrow N_n\sigma$ and $Q_1(\tilde{P}_1\sigma), \ldots, Q_m(\tilde{P}_m\sigma)$ then $M\sigma \Downarrow N\sigma$. Note that syntactically there is no difference between inference rules for big-step semantics and small-step semantics. The only difference is that in the small-step semantics, $\to$ is expected to denote one computation step while in the big-step semantics, $\Downarrow$ is expected to relate configurations to their associated final configuration.

Example 4. Continuing the previous examples, we consider the following big-step semantics $B = \{T_1, T_2\}$ for the call-by-value lambda calculus:

$$(T_1)\ \frac{}{X \Downarrow X}\ \mathrm{X}{\downarrow} \qquad (T_2)\ \frac{X \Downarrow \lambda x.X', Y \Downarrow Y', Z \Downarrow V}{XY \Downarrow V}\ \mathit{Subst}(X', x, Y', Z)$$

As for the small-step semantics, in the above rules $x, X, X', Y, Y', Z, V \in \mathcal{X}$ are variables and $x \in \mathcal{X}$ is sorted to be instantiated only with lambda-calculus variables $x_i \in \Sigma$. □

As a sanity check, it is expected for any big-step semantics that $M \Downarrow N$ implies $N{\downarrow}$. This will indeed be the case for the big-step semantics that are obtained by the algorithm that we describe next. In the following, we will write $\to^*$ for the reflexive-transitive closure of $\to$. The following definition captures the fact that a small-step semantics and a big-step semantics define the same programming language.

Definition 1. *A small-step semantics $\to$ and a big-step semantics $\Downarrow$ are* equivalent *when $M \to^* N$ and $N{\downarrow}$ hold if and only if $M \Downarrow N$ holds.*

The two semantics that we have defined above for call-by-value lambda calculus are equivalent (see, e.g., [13]):

Theorem 1. *The small-step semantics $\to$ defined by S in Example 3 is equivalent to the big-step semantics $\Downarrow$ defined by B in Example 4.*

Ideally, the big-step semantics and small-step semantics of a language should be equivalent in the sense of the definition above. However, producing a big-step semantics completely equivalent to the small-step semantics is sometimes impossible because, e.g., of non-determinism which can be described by small-step semantics but cannot be handled by big-step semantics (see discussion of non-determinism in Section 1). In such cases, it is desirable to have a slightly weaker link between the big-step semantics and the small-step semantics:

Definition 2. *A big-step semantics $\Downarrow$ is* sound *for a small-step semantics $\to$ if $M{\Downarrow}N$ implies $M{\to^*}N$ and $N{\downarrow}$.*

Note that if a small-step semantics $\to$ is equivalent to a big-step semantics $\Downarrow$, then it immediately follows that $\Downarrow$ is sound for $\to$. In all of the examples that we discuss in the rest of the paper, we will obtain big-step semantics that are fully equivalent to the initial small-step semantics. However, as discussed in Section 1, this cannot be the case for all languages. In such cases, it is desirable to establish that the two semantics satisfy the link in Definition 2.

3 From Small-Step Semantics to Big-Step Semantics

This section describes the transformation from small-step semantics to big-step semantics. We also give the class of small-step semantics for which our transformation is sound and complete in the sense of obtaining big-step semantics equivalent to the original small-step semantics.

3.1 The Transformation

The first idea that comes to mind when transforming a small-step semantics into a big-step semantics is to just add an explicit Transitivity-like inference rule. However, this defeats the purpose of having big-step semantics in the first case, since the $\to$ relation still explicitly appears in the inference system and must be reasoned about. Therefore, another approach is desirable.

Let $S = \{R_1, \ldots, R_k\}$ be a set of small-step inference rules $R_1, \ldots, R_k$ defining a small-step semantics. To S we associate the set $B(S) = \{R, R'_1, \ldots, R'_k\}$, where

$$R = \frac{}{V{\Downarrow}V}\ V{\downarrow}$$

and where

$$R'_i = \frac{M_1{\Downarrow}N_1, \ldots, M_n{\Downarrow}N_n, N{\Downarrow}V}{M{\Downarrow}V}\ Q_1(\tilde{P}_1), \ldots, Q_m(\tilde{P}_m)$$

for every inference rule

$$R_i = \frac{M_1 \to N_1, \ldots, M_n \to N_n}{M \to N} \; Q_1(\tilde{P}_1), \ldots, Q_m(\tilde{P}_m)$$

in S ($1 \leq i \leq k$). In the above rules, $V \in \mathcal{X}$ is a variable, $M, N, M_1, \ldots, M_n$, $N_1, \ldots, N_n$ are configurations, $\tilde{P}_1, \ldots, \tilde{P}_m$ are sequences of configurations and $Q_1, \ldots, Q_m$ are predicates.

Example 5. Continuing Example 3, we have that $B(S) = \{R, R_1', R_2', R_3'\}$ is:

$$R = \frac{}{V \Downarrow V} \; V{\downarrow} \qquad R_1' = \frac{X \Downarrow X', X'Y \Downarrow V}{XY \Downarrow V} \qquad R_2' = \frac{Y \Downarrow Y', XY' \Downarrow V}{XY \Downarrow V} \; X{\downarrow}$$

$$R_3' = \frac{Z \Downarrow V}{(\lambda x.X)Y \Downarrow V} \; Y{\downarrow}, \mathit{Subst}(X, x, Y, Z)$$

□

Note that for the call-by-value lambda calculus that we have used as a running example, the big-step semantics $B(S) = \{R, R_1', R_2', R_3'\}$ obtained automatically from the small-step semantics $S = \{R_1, R_2, R_3\}$ by the transformation described above is slightly different from the manually designed big-step semantics $B = \{T_1, T_2\}$ (defined in Example 4). The difference is that the automatically generated rules R_1', R_2', R_3' are synthesized into a single rule T_2 in the manually designed big-step semantics. It is not a surprise that the manually designed rules are slightly simpler than the automatically generated rules since the automated rules must be more generic. Note however that there is no redundancy in the generated rules and that the implementations of interpreters based on the two sets of rules would look very similar. We speculate that simplification rules could reduce the gap between the generated rules and the manually designed rules but we leave this as an open problem for further study.

3.2 The Assumptions

In order for the automatic derivation of the big-step semantics from the small-step semantics to produce a completely *equivalent semantics*, we require that the inference system S satisfies four assumptions. The big-step semantics are *sound* for the small-step semantics in the sense of Definition 2 when one of the four assumptions holds. In order to state the four assumptions, we need to notions of *star-soundness* and *star-completeness*, defined below.

Definition 3 (Star-sound Inference Rule). *A small-step inference rule*

$$R = \frac{M_1 \to N_1, \ldots, M_n \to N_n}{M \to N} \; Q_1(\tilde{P}_1), \ldots, Q_m(\tilde{P}_m)$$

is star-sound *if it still holds when $\to$ is replaced by $\to^*$, i.e. for any σ such that $M_1\sigma \to^* N_1\sigma, \ldots, M_n\sigma \to^* N_n\sigma$ and $Q_1(\tilde{P}_1\sigma), \ldots, Q_m(\tilde{P}_m\sigma)$ we have $M\sigma \to^* N\sigma$.*

Intuitively, star-soundness means that if one can take zero or more steps to reach N_i from M_i (for all $1 \leq i \leq n$), then M can also be reached from N in zero or more steps.

Definition 4 (Star-complete Inference Rule). *We say that a small-step inference rule*

$$R = \frac{M_1 \rightarrow N_1, \ldots, M_n \rightarrow N_n}{M \rightarrow N} \; Q_1(\tilde{P}_1), \ldots, Q_m(\tilde{P}_m)$$

is star-complete *w.r.t to a small-step semantics* $\rightarrow$ *if for every substitution* σ *such that* $M_1\sigma \rightarrow N_1\sigma, \ldots, M_n\sigma \rightarrow N_n\sigma, Q_1(\tilde{P}_1\sigma), \ldots, Q_m(\tilde{P}_m\sigma)$ *and* $N\sigma \rightarrow^* V$ *for some ground configuration* V *with* $V\!\downarrow$*, we have that there exists a substitution* σ' *which agrees with* σ *on* $\mathcal{V}ar(M, M_1, \ldots, M_n)$ *such that* $N_1\sigma \rightarrow^* N_1\sigma', \ldots, N_n\sigma \rightarrow^* N_n\sigma', Q_1(\tilde{P}_1\sigma'), \ldots, Q_m(\tilde{P}_m\sigma')$ *and* $N_1\sigma'\!\downarrow, \ldots, N_n\sigma'\!\downarrow$ *where the number of steps in each of the derivations* $N_i\sigma \rightarrow^* N_i\sigma'$ *(*$1 \leq i \leq n$*) is strictly smaller than the number of rewrite steps in the derivation* $N\sigma \rightarrow^* V$*.*

Intuitively, star-completeness means that if the rule can be used to start a terminating computation, then one can find terminating computations starting with M_i $(1 \leq i \leq n)$ as well. We are now ready to state our assumptions.

Assumption 1 (Ground Confluency). *The relation* $\rightarrow$ *induced by* S *is ground confluent: If* $M \rightarrow^* N_1$ *and* $M \rightarrow^* N_2$ *for some ground configurations* M, N_1, N_2*, then there exists a ground configuration* P *such that* $N_1 \rightarrow^* P$ *and* $N_2 \rightarrow^* P$*.*

Assumption 2. *For any ground configuration* M*, we have that* $M\!\downarrow$ *implies* $M \not\rightarrow N$ *for any ground configuration* N*.*

Assumption 3 (Star-soundness). *Any inference rule* $R \in S$ *is* star-sound *with respect to the rewrite relation* $\rightarrow$ *induced by* S*.*

Assumption 4 (Star-completeness). *Any inference rule* $R \in S$ *is* star-complete *with respect to the rewrite relation* $\rightarrow$ *induced by* S*.*

Our next theorem states that the transformation that we have presented is sound and complete in the sense that the resulting big-step semantics is equivalent to the original small-step semantics whenever S satisfies the above assumptions.

Theorem 2. *Let* $\rightarrow$ *be the small-step relation defined by* S *and let* $\Downarrow$ *be the big-step relation defined by* $B(S)$*. If* S *satisfies Assumptions 1, 2, 3, 4 defined in Subsection 3.2, then* $\rightarrow$ *and* $\Downarrow$ *are equivalent.*

Proof (Sketch). In one direction, the proof follows by induction on the number of small-steps taken and in the reverse direction by induction on the big-step proof tree.

The CBV lambda calculus in Example 3 satisfies the above assumptions:

Lemma 1. *The small-step semantics* $S = \{R_1, R_2, R_3\}$ *defined in Example 3 satisfies Assumptions 1, 2, 3, 4.*

Proof (Sketch.). It is well known (e.g., starting with the seminal result of Plotkin [13]) that Assumption 1 (confluence or "the Church-Rosser" property) holds for various (extensions of) lambda-calculi. Assumptions 2, 3, 4 follow by case analysis and induction.

Therefore we obtain immediately from Lemma 1 and Theorem 2:

Corollary 1. *The big-step semantics $\Downarrow$ defined by $B(S)$ in Example 5 is equivalent to the small-step semantics $\rightarrow$ defined by S in Example 3.*

It might seem that Assumptions 1, 2, 3, 4 are excessive. However, note that having these assumptions establishes a very strong link between the two semantics. If the big-step semantics should just be a sound approximation of the small-step semantics (i.e., when some behaviors of the small-step semantics can be discarded), then only Assumption 3 (star-soundness) is needed:

Theorem 3. *Let $\rightarrow$ be the small-step relation defined by S and let $\Downarrow$ be the big-step relation defined by $B(S)$. If S satisfies Assumption 3 defined in Subsection 3.2, then $\Downarrow$ is sound for $\rightarrow$ in the sense of Definition 2.*

Proof (sketch). By induction on the big-step proof tree.

4 Examples

4.1 Call-by-Name Lambda Calculus

We have already shown how our transformation works for CBV lambda calculus as a running example in Section 3. We consider the same signature as for CBV lambda calculus and the following set $S = \{R_1, R_2\}$ of small-step inference rules modeling call-by-name lambda calculus (CBN lambda calculus):

$$(R_1)\ \frac{X \rightarrow X'}{XY \rightarrow X'Y} \qquad\qquad (R_2)\ \frac{}{(\lambda x.X)Y \rightarrow Z}\ \mathit{Subst}(X, x, Y, Z)$$

The CBN big-step semantics obtained from the above definition is the set $B(S) = \{R, R_1', R_2'\}$, where:

$$(R_1')\ \frac{X \Downarrow X',\ X'Y \Downarrow V}{XY \Downarrow V}\ V{\downarrow} \qquad\qquad (R_2')\ \frac{Z \Downarrow V}{(\lambda x.X)Y \Downarrow V}\ \mathit{Subst}(X, x, Y, Z)$$

It can be shown that the CBN lambda-calculus defined above satisfies Assumptions 1, 2, 3 and 4 and therefore the above transformation is sound and complete.

4.2 Call-by-Value Mini-ML

Mini-ML is a folklore language used for teaching purposes which extends lambda-calculus with some features like numbers, booleans, pairs, let-bindings or fix-points in order to obtain a language similar to (Standard) ML. We use a variant of Mini-ML where the abstract syntax is:

$\mathsf{Var} ::=$	variables
$\mid x_0 \mid \ldots \mid x_n \mid \ldots$	
$\mathsf{Exp} ::=$	expressions
$\mid \mathsf{Var}$	variable
$\mid 0 \mid 1 \mid \ldots \mid n \mid \ldots$	natural number
$\mid \mathsf{Exp} + \mathsf{Exp}$	arithmetic sum
$\mid \lambda\mathsf{Var}.\mathsf{Exp}$	function definition
$\mid \mu\mathsf{Var}.\mathsf{Exp}$	recursive definition
$\mid \mathsf{Exp}\ \mathsf{Exp}$	function application
$\mid \mathsf{let}\ \mathsf{Var} = \mathsf{Exp}\ \mathsf{in}\ \mathsf{Exp}$	let binding

Here Exp and Var are sorts in the signature Σ, with Var being a subsort of Exp. The additional syntax can of course be desugared into pure lambda calculus, but we prefer to give its semantics directly in order to show how our transformation works. We define configurations to consist of expressions Exp and final configurations to be natural numbers $0, 1, \ldots, n, \ldots$ or function definitions $\lambda\mathsf{Var}.\mathsf{Exp}$. We consider the predicate $+(M, N, P)$ which holds when M, N and P are integers such that P is the sum of M and N. We define the small-step semantics of the language to be $S = \{R_1, \ldots, R_{10}\}$, where:

$$R_1 = \frac{X \rightarrow X'}{X + Y \rightarrow X' + Y} \qquad R_2 = \frac{Y \rightarrow Y'}{X + Y \rightarrow X + Y'}\, X\!\downarrow \qquad R_3 = \frac{}{X + Y \rightarrow Z}\, +(X, Y, Z)$$

$$R_4 = \frac{}{\mu x.X \rightarrow Z}\, \mathit{Subst}(X, x, \mu x.X, Z) \qquad R_5 = \frac{X \rightarrow X'}{XY \rightarrow X'Y} \qquad R_6 = \frac{Y \rightarrow Y'}{XY \rightarrow XY'}\, X\!\downarrow$$

$$R_7 = \frac{}{(\lambda x.X)Y \rightarrow Z}\, Y\!\downarrow, \mathit{Subst}(X, x, Y, Z) \qquad R_8 = \frac{X \rightarrow X'}{\mathsf{let}\ x = X\ \mathsf{in}\ Y \rightarrow \mathsf{let}\ x = X'\ \mathsf{in}\ Y}$$

$$R_9 = \frac{Y \rightarrow Y'}{\mathsf{let}\ x = X\ \mathsf{in}\ Y \rightarrow \mathsf{let}\ x = X\ \mathsf{in}\ Y'}\, X\!\downarrow$$

$$R_{10} = \frac{}{\mathsf{let}\ x = X\ \mathsf{in}\ Y \rightarrow Z}\, X\!\downarrow, Y\!\downarrow, \mathit{Subst}(Y, x, X, Z)$$

Rules R_1, R_2, R_3 describe integer arithmetic where the arguments to the plus operator are evaluated in order. Rule R_4 describes recursive definitions. The term $\mu x.M$ reduces to M where x is replaced by $\mu x.M$. This allows the definition of recursive functions. Note that $\mu x.M$ is not a final configuration. The next rules R_5, R_6, R_7 are those from the standard call-by-value lambda calculus and handle function application. Finally, rules R_8, R_9, R_{10} handle let bindings. Not surprisingly, the small-step semantics of Mini-ML satisfies Assumptions 1, 2, 3 and 4 as well. Therefore, by our result, the big-step semantics obtained by our transformation is equivalent to the small-step semantics.

4.3 IMP

Much like Mini-ML, IMP is a simple language used for teaching purposes. However, IMP is imperative and it usually features arithmetic and boolean expressions, variables, and statements such as assignment, conditionals and while-loops.

We define a variant of IMP with the following abstract syntax:

Var ::=	$\mathsf{variables}$	
	$\mid x_0 \mid \ldots \mid x_n \mid \ldots$	
Exp ::=	$\mathsf{expressions}$	
	$\mid \mathsf{Var}$	variable
	$\mid 0 \mid 1 \mid \ldots \mid n \mid \ldots$	natural number
	$\mid \mathsf{Exp} + \mathsf{Exp}$	arithmetic sum
	$\mid \mathsf{Exp} \leq \mathsf{Exp}$	comparison
Seq ::=	$\mathsf{sequence\ of\ statements}$	
	$\mid \mathsf{emp}$	empty sequence
	$\mid \mathsf{Var} := \mathsf{Exp}; \mathsf{Seq}$	assignment
	$\mid \mathsf{if\ Exp\ then\ Seq\ else\ Seq}; \mathsf{Seq}$	conditional
	$\mid \mathsf{while\ Exp\ do\ Seq}; \mathsf{Seq}$	loop
Pgm ::=	$\mathsf{programs}$	
	$\mid \mathsf{Seq}; \mathsf{Exp}$	execute statement return expression

For simplicity, we do not model booleans and we assume a C-like interpretation of naturals as booleans: any non-zero value is interpreted as truth and the comparison operator $\leq$ returns 0 (representing false) or 1 (representing true). We will define therefore the predicates $\mathit{Zero}(n)$ and $\mathit{NonZero}(n)$ which hold when n is a natural number equal to 0 (for *Zero*) and respectively when n is a natural number different from 0 (for *NonZero*).

Programs are described by terms of sort Pgm. As opposed to the previous examples of programming languages (all based on lambda-calculus), IMP programs do not run standalone; an IMP program runs in the presence of an *environment* which maps variables to natural numbers. Therefore the small-step relation will relate *configurations* which consist of a program and an environment:

Env ::=	$\mathsf{environment\ (list\ of\ bindings)}$	
	$\mid \emptyset$	empty
	$\mid \mathsf{Var} \mapsto \mathsf{Nat}, \mathsf{Env}$	non-empty
Cfg ::=	$\mathsf{configuration}$	
	$\mid (\mathsf{Pgm}, \mathsf{Env})$	program + environment

As environments are essentially defined to be lists of pairs $x \mapsto n$ (for variables x and naturals n), there is no stopping a variable from appearing twice in an environment (making the environment map the same variable to potentially different natural numbers). We break ties by making the assumption that the binding which appears first in a list for a given variable is the right one. We define the predicate $\mathit{Lookup}(e, x, n)$ to hold exactly when the variable x is mapped to the integer n by the environment e (breaking ties as described above). The predicate $\mathit{Update}(e, x, n, e')$ holds when e' is the environment obtained from e by letting x map to n.

A final configuration (at which the computation stops) is a configuration in which the program is the empty sequence of statements (emp) followed by a fully evaluated expression (i.e., a natural number). Therefore the predicate $\downarrow$ will be true exactly of configurations of the form $(\mathsf{emp}; n, e)$ where e is any environment and n is a natural number. To simplify notations, when the sequence of statements is empty, we also write (N, e) instead of $(\mathsf{emp}; N, e)$.

To define our semantics, we also consider a predicate $Nat(N)$ which holds exactly when N is a natural number, a predicate $+(M, N, P)$ which relates any two natural number M and N to their sum P, a predicate $\leq (M, N, P)$ which is true when M, N are natural numbers and $M \leq N$ implies $P = 1$ and $M > N$ implies $P = 0$. Then the small-step semantics of IMP is given by $S = \{R_1, \ldots, R_{10}\}$, where the rules $R_1, \ldots R_{10}$ are described below. The rules for evaluating expressions are the following (recall that (M, e) is short for $(\mathsf{emp}; M, e)$):

$$R_1 = \frac{(X, e) \to (X', e)}{(X + Y, e) \to (X' + Y, e)} \qquad R_2 = \frac{(Y, e) \to (Y', e)}{(X + Y, e) \to (X + Y', e)}\ Nat(X)$$

$$R_3 = \frac{}{(X + Y, e) \to (Z, e)}\ +(X, Y, Z) \qquad R_4 = \frac{}{(x, e) \to (Y, e)}\ Lookup(e, x, Y)$$

Assignments work by first evaluating the expression and then updating the environment:

$$R_5 = \frac{(X, e) \to (X', e)}{((x := X); Z; Y, e) \to ((x := X'); Z; Y, e)}$$

$$R_6 = \frac{}{((x := X); Z; Y, e) \to (Z; Y, e')}\ Nat(X),\ Update(e, x, X, e')$$

Note that, in the last two rules above, Z matches the remaining sequence of statements while Y matches the expression representing the result of the program. The rules for the conditional and for the while loop are as expected:

$$R_7 = \frac{(X, e) \to (X', e)}{\begin{array}{c}(\mathsf{if}\ X\ \mathsf{then}\ Y_1\ \mathsf{else}\ Y_2; Z; Y, e) \to \\ (\mathsf{if}\ X'\ \mathsf{then}\ Y_1\ \mathsf{else}\ Y_2; Z; Y, e)\end{array}} \qquad R_8 = \frac{}{\begin{array}{c}(\mathsf{if}\ X\ \mathsf{then}\ Y_1\ \mathsf{else}\ Y_2; Z; Y, e) \to \\ (Y_1; Z; Y, e)\end{array}}\ Zero(X)$$

$$R_9 = \frac{}{\begin{array}{c}(\mathsf{if}\ X\ \mathsf{then}\ Y_1\ \mathsf{else}\ Y_2; Z; Y, e) \to \\ (Y_2; Z; Y, e)\end{array}}\ NonZero(X)$$

$$R_{10} = \frac{}{\begin{array}{c}(\mathsf{while}\ X\ \mathsf{do}\ X_0; Z; Y, e) \to \\ (\mathsf{if}\ X\ \mathsf{then}\ (X_0; \mathsf{while}\ X\ \mathsf{do}\ X_0; \mathsf{emp}) \\ \mathsf{else}\ Z; \mathsf{emp}; Y, e)\end{array}}$$

Note that in the above 10 rules, $X, X', Y, Y', Z, x, e, X_0, Y_1, Y_2 \in \mathcal{X}$ are variables. Furthermore, e is sorted to only match environments. The big-step semantics $B(S) = \{R, R'_1, \ldots, R'_{10}\}$ obtained through our transformation is:

$$R = \frac{}{V \Downarrow V}\ V\!\downarrow \qquad R'_1 = \frac{(X, e) \Downarrow (X', e), (X' + Y, e) \Downarrow V}{(X + Y, e) \Downarrow V}$$

$$R'_2 = \frac{(Y, e) \Downarrow (Y', e), (X + Y', e) \Downarrow V}{(X + Y, e) \Downarrow V}\ Nat(X) \qquad R'_3 = \frac{(Z, e) \Downarrow V}{(X + Y, e) \Downarrow V}\ +(X, Y, Z)$$

$$R'_4 = \frac{(Y, e) \Downarrow V}{(x, e) \Downarrow V}\ Lookup(e, x, Y) \qquad R'_5 = \frac{(X, e) \Downarrow (X', e), ((x := X'); Z; Y, e) \Downarrow V}{((x := X); Z; Y, e) \Downarrow V}$$

$$R'_6 = \frac{(Z; Y, e')\Downarrow V}{((x := X); Z; Y, e)\Downarrow V} \; \mathit{Nat}(X), \mathit{Update}(e, x, X, e')$$

$$R'_7 = \frac{(X, e)\Downarrow(X', e), (\mathsf{if}\ X'\ \mathsf{then}\ Y_1\ \mathsf{else}\ Y_2; Z; Y, e)\Downarrow V}{(\mathsf{if}\ X\ \mathsf{then}\ Y_1\ \mathsf{else}\ Y_2; Z; Y, e)\Downarrow V}$$

$$R'_8 = \frac{(Y_1; Z; Y, e)\Downarrow V}{(\mathsf{if}\ X\ \mathsf{then}\ Y_1\ \mathsf{else}\ Y_2; Z; Y, e)\Downarrow V} \; \mathit{Zero}(X)$$

$$R'_9 = \frac{(Y_2; Z; Y, e)\Downarrow V}{(\mathsf{if}\ X\ \mathsf{then}\ Y_1\ \mathsf{else}\ Y_2; Z; Y, e)\Downarrow V} \; \mathit{NonZero}(X)$$

$$R'_{10} = \frac{(\mathsf{if}\ X\ \mathsf{then}\ (X_0; \mathsf{while}\ X\ \mathsf{do}\ X_0; \mathsf{emp})\ \mathsf{else}\ Z; \mathsf{emp}; Y, e)\Downarrow V}{(\mathsf{while}\ X\ \mathsf{do}\ X_0; Z; Y, e)\Downarrow V}$$

It can be shown that the small-step semantics of IMP, as defined above, satisfies Assumptions 1, 2, 3 and 4. Therefore, by our result, the big-step semantics described above is equivalent to the small-step semantics. Note that Assumption 1 (confluence) is satisfied due to the deterministic nature of the language.

5 Related Work

Each semantic style has its own advantages and disadvantages. Work on addressing the disadvantages of big-step semantics includes [14], which presents a method to reduce verbosity called *pretty big step semantics* and [10,15], proposing methods of distinguishing between non-terminating and stuck configurations. However, it is surprising that little work has gone into (automatic) transformation of one style of semantics into another, given that proving that two semantics are equivalent can be nontrivial.

Huizing *et al.* [16] show how to automatically transform big-step semantics into small-step semantics. In some sense, they propose the exact inverse of our transformation with the goal of obtaining a semantics suitable for reasoning about concurrency. However, their transformation is not natural in the sense that the small-step semantics does not look like what would be written "by hand": instead, a stack is artificially added to program configurations in order to obtain the small-step semantics.

A line of work by Danvy et al. ([17,18,19,20]) shows that *functional implementations* of small-step semantics and big-step semantics can be transformed into each other via sound program transformations (like the well-known CPS transform). Their transformation looks similar to ours, but operates on interpreters of the language and is different from our work in several significant ways. Firstly, we address the transformation of the *semantics* itself (defined in a simple meta-language) and not of the implementation of the semantics as an interpreter written in a functional language. This is in principle somewhat more powerful since tools other than interpreters (like program verifiers) can also take advantage of the newly obtained semantics. Secondly, because our transformation is completely formal, we are able to *prove* that it is correct (under a number of assumptions on the initial small-step semantics). This is in contrast to the above line of work, where the program transformations are performed manually and are shown to produce equivalent interpreters for a set of example languages. In particular, Danvy et al.

do not prove a meta-theorem stating that their transformation is sound for any language – they conclude that small-step machines can be mechanically transformed into big-step machines by generalizing from a set of example languages. On the other hand and in contrast to the present work, their transformation also works in reverse (obtaining small-step machines from big-step machines) and is more flexible than ours since the set of transformations is not fixed a-priori.

In order to formalize our transformation we define in Section 2 a *meta-language* for describing small-step (and big-step) structural operational semantics. Such meta-languages (also called *rule formats*) abound [21,22] in the literature. However these restricted rule formats are used to prove meta-theorems about well-definedness, compositionality, etc. and not for changing the style of the semantics.

6 Discussion and Further Work

We have presented a very simple syntactic trick for transforming small-step semantics into big-step semantics. We have analysed the transformations on several examples including both lambda-calculus based languages and an imperative language and we have identified four assumptions under which the transformation is sound and complete in the sense of yielding a big-step semantics which is equivalent to the initial small-step semantics. The confluence assumption (1) seems to be unavoidable since we have already shown that big-step semantics cannot deal with non-determinism; furthermore, it is already known for many variants and extensions of lambda-calculi that they satisfy confluence. The second assumption (2) regarding the definition of the final configuration ($\downarrow$) predicate does not seem to be too strong since it is what we expect of any sound small-step semantics. Finally, the last two assumptions (3 – star-soundness and 4 – star-completeness) are the most problematic in the sense that they are semantic assumptions which must be proven to hold. Note however that they hold for a variety of programming languages that we have analysed (imperative, functional) in different settings (call-by-name, call-by-value) and with both explicit (for the IMP language) and implicit substitutions (for the lambda-calculi). However, obtaining a sound syntactic approximation for the last two assumptions is an important step for further work.

We have also shown (Theorem 3) that a sound big-step semantics can be obtained from the initial small-step semantics under Assumption 3 (start-soundness) only. Having a sound (but not necessarily complete) big-step semantics can be acceptable in case the big-step semantics is used for generating a compiler, since a compiler is free to choose among the behaviors of the program. This could lead to obtaining an (efficient) compiler directly from the small-step semantics. As future work, we would like to investigate syntactic approximations of the four assumptions, the degree to which checking the assumptions can be automated and the possibility of generating variations of big-step semantics such as the ones in [10,14,23].

References

1. Strachey, C.: Towards a formal semantics. In: Formal Language Description Languages for Computer Programming, pp. 198–220. North-Holland (1966)

2. Floyd, R.W.: Assigning meanings to programs. In: Schwartz, J.T. (ed.) Mathematical Aspects of Computer Science, pp. 19–32. AMS, Providence (1967)
3. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
4. Plotkin, G.D.: A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus (1981)
5. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. 60-61, 17–139 (2004)
6. Kahn, G.: Natural semantics. In: Brandenburg, F.J., Wirsing, M., Vidal-Naquet, G. (eds.) STACS 1987. LNCS, vol. 247, pp. 22–39. Springer, Heidelberg (1987)
7. Mosses, P.D.: Modular structural operational semantics. J. Log. Algebr. Program. 60-61, 195–228 (2004)
8. Pettersson, M.: A compiler for natural semantics. In: Gyimóthy, T. (ed.) CC 1996. LNCS, vol. 1060, pp. 177–191. Springer, Heidelberg (1996)
9. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine, and compiler. ACM T. Prog. Lang. Syst. 28, 619–695 (2006)
10. Leroy, X., Grall, H.: Coinductive big-step operational semantics. Information and Computation 207(2), 284–304 (2009)
11. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. Journal of Logic and Algebraic Programming 79(6), 397–434 (2010)
12. Meseguer, J., Roşu, G.: The rewriting logic semantics project: A progress report. In: Owe, O., Steffen, M., Telle, J.A. (eds.) FCT 2011. LNCS, vol. 6914, pp. 1–37. Springer, Heidelberg (2011)
13. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theor. Comput. Sci. 1(2), 125–159 (1975)
14. Charguéraud, A.: Pretty-big-step semantics. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems. LNCS, vol. 7792, pp. 41–60. Springer, Heidelberg (2013)
15. Cousot, P., Cousot, R.: Bi-inductive structural semantics. Information and Computation 207(2), 258–283 (2009)
16. Huizing, C., Koymans, R., Kuiper, R.: A small step for mankind. In: Dams, D., Hannemann, U., Steffen, M. (eds.) de Roever Festschrift. LNCS, vol. 5930, pp. 66–73. Springer, Heidelberg (2010)
17. Danvy, O.: Defunctionalized interpreters for programming languages. In: ICFP 2008, pp. 131–142. ACM, New York (2008)
18. Danvy, O., Millikin, K.: On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. Inf. Process. Lett. 106(3), 100–109 (2008)
19. Danvy, O., Millikin, K., Munk, J., Zerny, I.: Defunctionalized interpreters for call-by-need evaluation. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 240–256. Springer, Heidelberg (2010)
20. Danvy, O., Johannsen, J., Zerny, I.: A walk in the semantic park. In: Khoo, S.C., Siek, J.G. (eds.) PEPM, pp. 1–12. ACM (2011)
21. Groote, J.F., Mousavi, M., Reniers, M.A.: A hierarchy of SOS rule formats. In: Proceedings of SOS 2005, Lisboa, Portugal. ENTCS, Elsevier (2005)
22. Aceto, L., Fokkink, W., Verhoef, C.: Structural operational semantics. In: Handbook of Process Algebra, pp. 197–292. Elsevier (1999)
23. Uustalu, T.: Coinductive big-step semantics for concurrency. In: Vanderbauwhede, W., Yoshida, N. (eds.) Proceedings of PLACES 2013. EPTCS (2013) (to appear)

Program Equivalence by Circular Reasoning

Dorel Lucanu[1] and Vlad Rusu[2]

[1] Al. I. Cuza University of Iaşi, Romania
dlucanu@info.uaic.ro
[2] Inria Lille Nord-Europe, France
Vlad.Rusu@inria.fr

Abstract. We propose a logic and a deductive system for stating and automatically proving the equivalence of programs in deterministic languages having a rewriting-based operational semantics. The deductive system is circular in nature and is proved sound and weakly complete; together, these results say that, when it terminates, our system correctly solves the program-equivalence problem as we state it. We show that our approach is suitable for proving the equivalence of both terminating and non-terminating programs, and also the equivalence of both concrete and symbolic programs. The latter are programs in which some statements or expressions are symbolic variables. By proving the equivalence between symbolic programs, one proves in one shot the equivalence of (possibly, infinitely) many concrete programs obtained by replacing the variables by concrete statements or expressions. A prototype of the proof system for a particular language was implemented and can be tested on-line.

1 Introduction

In this paper we propose a formal notion of program equivalence, together with a logic for expressing this notion and a deductive system for automatically proving it. Programs can belong to any deterministic language whose semantics is specified by a set of rewrite rules. The equivalence we consider is a form of weak bisimulation, allowing several instructions of one program to be matched by several instructions of the other one. The proof system is circular: its conclusions can be re-used as hypotheses in a controlled way. It is not guaranteed to terminate, but when it does terminate, our proof system correctly solves the program-equivalence problem as stated, thanks to its soundness and weak completeness properties. These are informally presented below and are formalised and proved in the paper.

The proposed framework is also suitable for proving the equivalence of *symbolic programs.* These are programs in which some expressions and/or statements are *symbolic variables*, which denote sets of concrete programs obtained by substituting the symbolic variables by concrete expressions and/or statements. Thus, by proving the equivalence between symbolic programs, one proves in just one shot the equivalence of (possibly, infinitely) many concrete programs, which has applications in the verification of certain classes of compilers/translators. Here is an example of equivalent symbolic programs.

Example 1. Assume that we want to translate between a language that has `for`-loops into a language that only has `while`-loops. This amounts to translating the symbolic program in the left-hand side to the one in the right-hand side.

`for` I `from` A `to` B `do{` S `}` $\qquad$ I `=` A `;while` I `<=` B `do {` S `;` I `=` I `+ 1 }`

Their symbolic variables I, A, B, S can be matched by, respectively, any identifier (I), arithmetical expression (A, B), and program statement (S). If we prove the equivalence between these two symbolic programs (as we shall do in this paper as an illustrative example) then we also prove that every concrete instance of the `for`-loop is equivalent to its translation to a concrete `while`-loop (or vice-versa). Nonterminating programs can be proved equivalent as well, e.g. by replacing the test I `<=` B with `not(`I `=` B`)` and by assuming nonterminating `for` loops when $A > B$, some instances of the above two symbolic programs are nonterminating.

In the rest of the paper we often refer to symbolic programs just as "programs".

A typical use of our program-equivalence framework consists in:

1. defining the operational semantics of a programming language, say, $\mathcal{L}$;
2. defining a language $\mathcal{L}^{sym}$, which extends the syntax of and semantics of $\mathcal{L}$, such that the programs in $\mathcal{L}^{sym}$ are exactly the symbolic programs of $\mathcal{L}$;
3. applying our deductive system to check the equivalence of programs in $\mathcal{L}^{sym}$.

Running the deductive system amounts essentially to executing the semantics of $\mathcal{L}^{sym}$ on pairs of $\mathcal{L}^{sym}$-programs. This may lead to any of the following outcomes:

– termination with success, in which case the programs given as input to the deductive system are equivalent, due to the deductive system's *soundness*;
– termination with failure, in which case the programs given as input to the deductive system are not equivalent, due to the system's *weak completeness*;
– non-termination, in which case nothing can be concluded about equivalence.

Non-termination is inherent in any sound automatic system for proving program equivalence, because the equivalence problem is undecidable. We show, however, that our system terminates when the programs given to it as inputs terminate, and also when they do not terminate but behave in a certain regular way (by infinitely repeating pairs of so-called *observationally equivalent configurations*).

Contributions. A logic and a proof system suitable for stating and proving the equivalence of concrete and of symbolic programs, as well as that of terminating and non-terminating ones. Programs can be written in any deterministic language that has a formal operational semantics based on term rewriting. We prove the soundness and weak completeness of our proof system, which ensure that the system correctly solves the program equivalence problem as we state it. A prototype implementation of the proposed deductive system is also presented.

Related Work. An exhaustive bibliography on the program-equivalence problem is outside the scope of this paper, as this problem is even older than the program-verification problem. Among the recent works perhaps the closest to ours is [1]. They also deal with the equivalence of parameterised programs (symbolic, in our terminology) and define equivalence in terms of bisimulation.

Their approach is, however, very different from ours. One major difference lies in the models of programs: [1] use CFGs (control flow graphs) of programs, while we use the operational semantics of languages. CFGs are more restricted, e.g., they are not well adapted to recursive or object-oriented programs, whereas operational semantics do not have these limitations. Of course, our advantage will only become apparent when we actually apply our approach to such programs.

Other closely related recent works are [2,3,4]. The first one targets programs that include recursive procedures, the second one exploits similarities between single-threaded programs in order to prove their equivalence, and the third one extends the latter to multi-threaded programs. They use operational semantics (of a specific language) and proof systems, and formally prove their proof system's soundness. In [2] they make a useful classification of equivalence relations used in program-equivalence research, and use these relations in their work.

However, all the relations classified in [2] are of an input/output nature: for given (sequences of) inputs, programs generate equal (sequences of) outputs and/or do not terminate. Such relations are well adapted for concrete programs with inputs and outputs, but not to symbolic programs with symbolic statements, for which a clear input-output relation may not exist. Indeed, symbolic statements may denote arbitrary concrete statements - including ones that do not perform input/output - actually, when symbolic programs are concerned, one cannot even rely on the existence of inputs and outputs. One may rely, however, on the observations of the effects of symbolic statements on the program's environment (e.g., values of variables). Our notion of weak bisimulation (up to a certain observation relation) allows this, both for finitely and for infinitely many repeated observations. We also show that some of the relations from [2] can be encoded in our relation by adding information to the program environment.

Many works on program equivalence arise from the verification of compilation in a broad sense. At one end there is full compiler verification [5], and at the other end, the so-called translation validation, i.e., the individual verification of each compilation [6] (we only cite two of the most relevant recent works). As also observed by [1], symbolic program verification can also be used for certain compilers, in which one proves the equivalence of each basic instruction pattern from the source language with its translation in the target language. The application of this observation to the verification of a compiler (from another project we are involved in) is ongoing and will be presented in another paper.

Several other works have targeted specific classes of languages: functional [7], microcode [8], CLP [9]. In order to be less language-specific some works advocate the use of intermediate languages, such as [10], which works on the Boogie intermediate language. And finally, only a few approaches, among which [5,8], deal with real-life language and industrial-size programs in those languages. This is in contrast to the equivalence checking of hardware circuits, which has entered the mainstream industrial practice (see, e.g., [11] for a survey on this topic).

Our proof system is inspired by that of *circular coinduction* [12], which allows one to prove equalities of data structures such as infinite streams and regular expressions. A notable difference between the present approach and [12] is that our specifications are essentially rewrite theories (meant to define the semantics of programming languages), whereas those of [12] are behavioural equational theories, a special class of equational specifications with visible and hidden sorts.

The rest of the paper is organised as follows. Section 2 presents our running example: IMP, a simple imperative language and its definition in $\mathbb{K}$ [13]. $\mathbb{K}$ is a formal framework for defining operational semantics of programming languages.

Our approach is, however, independent of the $\mathbb{K}$ framework and the IMP language; hence, we present a general, abstract notion of language definition in Section 3, and show how the $\mathbb{K}$ definition of IMP is an instance of that notion.

Section 4 contains our proposed definition for program equivalence, and Section 5 gives the syntax and semantics of a logic capturing the chosen equivalence.

Section 6 introduces two operations on formulas of the logic (derivatives and conjunction) which are used in our circular proof system for formula validity.

The proof system itself is presented in Section 7, together with its soundness and weak completeness results. The results say that, when it terminates, the proof system correctly answers to the question of whether its input (which is a set of formulas in our program-equivalence logic) denotes equivalent programs.

The conclusion and future work are presented in Section 8. Finally, formal proofs of the results in the paper are given in the technical report that can be found at `http://hal.archives-ouvertes.fr/hal-00744374/` .

2 A Simple Imperative Language and Its Semantics in $\mathbb{K}$

The language we are using as running example is IMP, a simple imperative language intensively used in research papers. A full $\mathbb{K}$ definition of it can be found in [13]. The syntax of IMP is described in Figure 1 and is mostly self-explained. The attribute (given as an annotation) *strict* from the syntax means the arguments of the annotated expression/statement are evaluated before the expression/statement itself is evaluated/executed. If the attribute has as arguments a list of natural numbers, then only the arguments in positions specified by the list are evaluated before the expression/statement. The *strict* attribute is actually syntactic sugar for a set of $\mathbb{K}$ rules, briefly presented later in the section.

The *configuration* of an IMP program consists of code to be executed and an enviroment mapping identifiers to integers. In $\mathbb{K}$, this is written as a nested structure of *cells*: here, a top cell `cfg`, having a cell `k` and a cell `env` (see Figure 2).

The cell `k` includes the code to be executed, represented as a list of computation tasks $C_1 \curvearrowright C_2 \curvearrowright \ldots$, meaning that first C_1 will be executed, then C_2, etc. Computation tasks are typically the evaluation of statements and expressions. The cell `env` is an environment that binds the program variables to values; such a binding is written as a multiset of bindings of the form, e.g., `a` $\mapsto 3$.

The semantics of IMP is given by a set of rules (see Figure 3) that say how the configuration evolves when the first computation task (statement or instruction) from the `k` cell is executed. The dots in a cell mean that the rest of the cell remains unchanged. Except for the conjunction, negation, and `if` statement, the semantics of each operator and statement is described by exactly one rule.

$$\begin{array}{lll}
\mathit{Int} & ::= & \text{domain of integer numbers (including operations)}\\
\mathit{Bool} & ::= & \text{domain of boolean constants (including operations)}\\
\mathit{Id} & ::= & \text{domain of identifiers}
\end{array}$$

$$\begin{array}{llll}
\mathit{AExp} ::= \mathit{Int} \mid \mathit{Id} & & \mathit{BExp} ::= \mathit{Bool} & \\
\quad \mid \mathit{AExp}\ \texttt{/}\ \mathit{AExp}\ [\text{strict}] & & \quad \mid \mathit{AExp}\ \texttt{<=}\ \mathit{AExp}\ [\text{strict}] & \\
\quad \mid \mathit{AExp}\ \texttt{*}\ \mathit{AExp}\ [\text{strict}] & & \quad \mid \texttt{not}\ \mathit{BExp}\ [\text{strict}] & \\
\quad \mid \mathit{AExp}\ \texttt{+}\ \mathit{AExp}\ [\text{strict}] & & \quad \mid \mathit{BExp}\ \texttt{and}\ \mathit{BExp}\ [\text{strict}(1)] & \\
\quad \mid (\mathit{AExp}) & & \quad \mid (\mathit{BExp}) &
\end{array}$$

$$\begin{array}{ll}
\mathit{Stmt} ::= \texttt{skip} \mid \mathit{Stmt}\ \texttt{;}\ \mathit{Stmt} & \mid \texttt{\{}\ \mathit{Stmt}\ \texttt{\}}\\
\quad \mid \mathit{Id}\ \texttt{=}\ \mathit{AExp} & \mid \texttt{while}\ \mathit{BExp}\ \texttt{do}\ \mathit{Stmt}\\
\quad \mid \texttt{if}\ \mathit{BExp}\ \texttt{then}\ \mathit{Stmt} & \mid \texttt{for}\ \mathit{Id}\ \texttt{from}\ \mathit{AExp}\ \texttt{to}\ \mathit{AExp}\\
\qquad \texttt{else}\ \mathit{Stmt}\ [\text{strict}(1)] & \quad \texttt{do}\ \mathit{Stmt}\ [\text{strict}(2,3)]
\end{array}$$

$$\mathit{Code} ::= \mathit{Id} \mid \mathit{Int} \mid \mathit{Bool} \mid \mathit{AExp} \mid \mathit{BExp} \mid \mathit{Stmt} \mid \mathit{Code} \curvearrowright \mathit{Code}$$

Fig. 1. $\mathbb{K}$ Syntax of IMP

$$\mathit{Cfg} ::= \langle\langle \mathit{Code}\rangle_{\mathsf{k}}\langle \mathit{Map}\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$$

Fig. 2. $\mathbb{K}$ Configuration of IMP

In Figure 3, the operations $\mathit{lookup} : \mathit{Map} \times \mathit{Id} \to \mathit{Int}$ and $\mathit{update} : \mathit{Map} \times \mathit{Id} \times \mathit{Int} \to \mathit{Map}$ are part of the domain of maps and have the usual meanings: *lookup* returns the value of an identifier in a map, and *update* modifies the map by adding (or, if it exists, by updating) the binding of an identifier to a value.

In addition to the rules in Figure 3 there are rules induced by the strictness of some statements. For example, the `if` statement is strict only in the first argument, meaning that this argument is evaluated before the `if` statement. This amounts to the following rules (automatically generated by the $\mathbb{K}$ tool):

$$\langle\langle \texttt{if}\ BE\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle BE \curvearrowright \texttt{if}\ \square\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle B \curvearrowright \texttt{if}\ \square\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$

where BE ranges over $\mathit{BExp} \setminus \{\mathit{false}, \mathit{true}\}$, B ranges over $\{\mathit{false}, \mathit{true}\}$, and $\square$ is a special variable destined to receive the value of BE once it is computed.

3 A Generic Notion of Language Definition

Our program-equivalence approach is independent of the formal framework used for defining languages as well as from the languages being defined. We thus propose a general notion of language definition and illustrate it later in the section on the $\mathbb{K}$ definition of IMP. We assume the reader is familiar with the basics of algebraic specification and rewriting. A language $\mathcal{L}$ is defined by:

1. A many-sorted algebraic signature Σ, which includes at least a sort *Cfg* for configurations and a subsignature Σ^{Bool} for Booleans with their usual

$$\langle\langle I_1 \texttt{ + } I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I_1 +_{Int} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle I_1 \texttt{ * } I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I_1 *_{Int} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle I_1 \texttt{ / } I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \wedge I_2 \neq 0 \Rightarrow \langle\langle I_1 /_{Int} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle I_1 \texttt{ <= } I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I_1 \leq_{Int} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \mathit{true} \texttt{ and } B \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle B \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \mathit{false} \texttt{ and } B \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \mathit{false} \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{not } \mathit{true} \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \mathit{false} \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{not } \mathit{false} \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \mathit{true} \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{skip } \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle S_1 ; S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1 \curvearrowright S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{\{ } S \texttt{ \}} \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{if } \mathit{true} \texttt{ then } S_1 \texttt{ else } S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{if } \mathit{false} \texttt{ then } S_1 \texttt{ else } S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{while } B \texttt{ do } S \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow$$
$$\quad\langle\langle \texttt{if } B \texttt{ then\{ } S \texttt{ ;while } B \texttt{ do } S \texttt{ \}else skip} \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{for } X \texttt{ from } I_1 \texttt{ to } I_2 \texttt{ do } S \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow$$
$$\quad\langle\langle X \texttt{ = } I_1 \texttt{ ;if } X \texttt{ <= } I_2 \texttt{ then\{ } S \texttt{ ;for } X \texttt{ from } I_1 \texttt{ + } 1 \texttt{ to } I_2 \texttt{ do } S \texttt{ \}else skip} \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle X \cdots\rangle_{\mathsf{k}} \langle \mathit{Env}\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \mathit{lookup}(\mathit{Env}, X) \cdots\rangle_{\mathsf{k}} \langle \mathit{Env}\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle X \texttt{ = } I \cdots\rangle_{\mathsf{k}} \langle \mathit{Env}\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \cdots\rangle_{\mathsf{k}} \langle \mathit{update}(\mathit{Env}, X, I)\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}}$$

Fig. 3. $\mathbb{K}$ Semantics of IMP

constants and operations. Σ may also include other subsignatures for other data sorts, depending on the language $\mathcal{L}$ (e.g., integers, identifiers, lists, maps,...). Let Σ^{Data} denote the subsignature of Σ consisting of all data sorts and their operations. We assume that the sort *Cfg* and the syntax of $\mathcal{L}$ are not data, i.e., they are defined in $\Sigma \setminus \Sigma^{Data}$, and that terms of sort *Cfg* have exactly one subterm denoting statements (which are programs in the syntax of $\mathcal{L}$) remaining to be executed. Let T_Σ denote the Σ-algebra of ground terms and $T_{\Sigma,s}$ denote the set of ground terms of sort s. Given a sort-wise infinite set of variables *Var*, let $T_\Sigma(\mathit{Var})$ denote the free Σ-algebra of terms with variables, $T_{\Sigma,s}(\mathit{Var})$ denote the set of terms of sort s with variables, and $var(t)$ denote the set of variables occurring in the term t.

2. A Σ-algebra $\mathcal{T}$. Let $\mathcal{T}_s$ denote the elements of $\mathcal{T}$ that have the sort s; the elements of $\mathcal{T}_{Cfg}$ are called *configurations*. $\mathcal{T}$ interprets the data sorts (those included in the subsignature Σ^{Data}) according to some Σ^{Data}-algebra $\mathcal{D}$. $\mathcal{T}$ interprets the non-data sorts (statements) as ground terms over the signature

$$(\Sigma \setminus \Sigma^{Data}) \cup \bigcup_{d \in sorts(\Sigma^{Data})} \mathcal{D}_d \tag{1}$$

where $\mathcal{D}_d$ denotes the carrier set of the sort d in the algebra $\mathcal{D}$, and the elements of $\mathcal{D}_d$ are added to the signature $\Sigma \setminus \Sigma^{Data}$ as constants of sort

d. That is, a language is parametric in the way its data are implemented; it just assumes there is such an implementation Σ^{Data}. This is important for technical reasons (implementing unification by matching, discussed below). Any *valuation* $\rho : Var \to \mathcal{T}$ is extended to a (homonymous) Σ-algebra morphism $\rho : T_\Sigma(Var) \to \mathcal{T}$. The interpretation of a ground term t in $\mathcal{T}$ is denoted by $\mathcal{T}_t$. If $b \in T_{\Sigma,Bool}(Var)$ then we write $\rho \models b$ iff $\rho(b) = \mathcal{D}_{true}$. For simplicity, we often write in the sequel $true, false$ instead of $\mathcal{D}_{true}, \mathcal{D}_{false}$.

3. A set $\mathcal{S}$ of rewrite rules, whose definition is given later in the section.

We explain these concepts on the IMP example. Each nonterminal from the syntax ($Int, Bool, AExp, \ldots$) is a sort in Σ. Each production from the syntax defines an operation in Σ; for instance, the production $AExp ::= AExp$ `+` $AExp$ defines the operation `_+_` $: AExp \times AExp \to AExp$. These operations define the constructors of the result sort. For the configuration sort Cfg, the only constructor is $\langle\langle\_\rangle_{\mathsf{k}}\langle\_\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} : Code \times Map_{Id,Int} \to Cfg$. The expression $\langle\langle X \texttt{ = } I \curvearrowright C\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ is a term of $T_{Cfg}(Var)$, where X is a variable of sort Id, I is a variable of sort Int, C is a variable of sort $Code$ (the rest of the computation), and Env is a variable of sort $Map_{Id,Int}$ (the rest of the environment). The data algebra $\mathcal{D}$ interprets Int as the set of integers, the operations like $+_{Int}$ (cf. Figure 3) as the corresponding usual operation on integers, $Bool$ as the set of Boolean values $\{false, true\}$, the operation like $\wedge$ as the usual Boolean operations, the sort $Map_{Id,Int}$ as the multiset of maps $X \mapsto I$, where X ranges over identifiers Id and I over the integers. The fact that maps are modified only by the *update* operation ensures that each identifiers is bound to at most one integer value. The other sorts, $AExp$, $BExp$, $Stmt$, and $Code$, are interpreted in the algebra $\mathcal{T}$ as ground terms over a modification of the form (1) of the signature Σ, in which data subterms are replaced by their interpretations in $\mathcal{D}$. For instance, the term `if` $1 >_{Int} 0$ `then skip else skip` is interpreted in $\mathcal{T}$ as `if` $true$ `then skip else skip`, since $\mathcal{D}$ interprets $1 >_{Int} 0$ as $\mathcal{D}_{true}(= true)$.

Definition 1 (pattern [14]). *A* pattern *is an expression of the form* $\pi \wedge b$*, where* $\pi \in T_{\Sigma,Cfg}(Var)$ *are* basic patterns*,* $b \in T_{\Sigma,Bool}(Var)$*, and* $var(b) \subseteq var(\pi)$*. If* $\gamma \in \mathcal{T}_{Cfg}$ *and* $\rho : Var \to \mathcal{T}$ *we write* $(\gamma, \rho) \models \pi \wedge b$ *for* $\gamma = \rho(\pi)$ *and* $\rho \models b$*.*

A basic pattern π defines a set of (concrete) configurations, and the condition b gives additional constraints these configurations must satisfy. In [14] patterns are encoded as FOL formulas, hence the conjunction notation $\pi \wedge b$. In this paper we keep the notation but separate basic patterns from constraining formulas.

We identify basic patterns π with paterns $\pi \wedge true$. Examples of patterns are $\langle\langle I_1 \texttt{ + } I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ and $\langle\langle I_1 \texttt{ / } I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge I_2 \neq 0$.

Definition 2 (semantical rule and transition system). *A* rule *is a pair of patterns of the form* $l \wedge b \Rightarrow r$ *(note that* r *is the pattern* $r \wedge true$*). Any set* $\mathcal{S}$ *of rules defines a labelled transition system* $(\mathcal{T}_{Cfg}, \Rightarrow^{\mathcal{T}}_{\mathcal{S}})$ *such that* $\gamma \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma'$ *iff there are* $(l \wedge b \Rightarrow r) \in \mathcal{S}$ *and* $\rho : Var \to \mathcal{T}$ *such that* $(\gamma, \rho) \models l \wedge b$ *and* $(\gamma', \rho) \models r$*.*

A configuration γ is *final* if its program subterm is empty. A configuration γ is a *deadlock* if it is not final and there is no configuration γ' such that $\gamma \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma'$.

Deadlocks are erroneous program terminations, e.g., division-by-zero attempts. A language is *deterministic* if its transition system $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ is deterministic.

Assumption 1. *We assume that the transition system $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ is deterministic.*

We shall be using unification in our program-equivalence deductive system. We call *symbolic unifier* of two terms t_1, t_2 any substitution $\sigma : var(t_1) \uplus var(t_2) \to T_{\Sigma}(Z)$ for some set Z of variables such that $t_1\sigma = t_2\sigma$. We call a *concrete unifier* of terms t_1, t_2 any valuation $\rho : var(t_1) \uplus var(t_2) \to \mathcal{T}$ such that $t_1\rho = t_2\rho$.

Assumption 2. *For all rules $(l \wedge b \Rightarrow r) \in \mathcal{S}$ and all patterns $\pi \in T_{\Sigma,Cfg}(Var)$ with $var(l) \cap var(\pi) = \emptyset$, there is a finite, possibly empty set $U(\pi, l)$ of symbolic unifiers of π and l, which satisfy the property that for all concrete unifiers ρ of π and l, there exist substitutions $\sigma \in U(\pi, l)$ and valuations η such that $\sigma\eta = \rho$.*

In related work [15] we prove that the above assumption can always be satisfied, by implementing unification with the rules of $\mathcal{L}$ by the *matching* with the rules of a language $\mathcal{L}^{sym}$, which extends the definition of $\mathcal{L}$ such that the symbolic execution of programs in $\mathcal{L}$ is the usual execution of programs in $\mathcal{L}^{sym}$. We illustrate how this is done via an example; other examples follow in the paper.

Example 2. Consider the pattern $\langle\langle \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ of sort *Cfg*, where B is a variable of sort *Bool* and S_1, S_2 are variables of sort *Stmt*, and the rule $\langle\langle(\texttt{if } true \texttt{ then } S_1' \texttt{ else } S_2') \curvearrowright S\rangle_{\mathsf{k}}\langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1' \curvearrowright S\rangle_{\mathsf{k}}\langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$. Here we have filled in the "$\ldots$" from Figure 3 with actual variables, and the rule's variables were chosen so that they are distinct from those in the formula. Let π denote the basic pattern and l the left-hand side of the rule. The set $U(\pi, l)$ is a singleton given by the substitution $\sigma = (B \mapsto true, S_1' \mapsto S_1, S_2' \mapsto S_2, M' \mapsto M)$. On the other hand, l does not match π because the constant leaf *true* of l does not match the variable B in π. However, the rule can be equivalently rewritten as

$$\langle\langle(\texttt{if } B' \texttt{ then } S_1' \texttt{ else } S_2') \curvearrowright\rangle_{\mathsf{k}}\langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge B' = true \Rightarrow \langle\langle S_1' \curvearrowright S\rangle_{\mathsf{k}}\langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$$

and now, there is match between the configuration l' from the left-hand side of the new rule and π, i.e., $(B' \mapsto B, S_1' \mapsto S_1, S_2' \mapsto S_2, M' \mapsto M)$. This match, combined with the condition $B' = true$, amount to the above symbolic unifier σ.

4 Defining Program Equivalence

We define in this section our notion of program equivalence. We base our definition on the transition system $(\mathcal{T}_{Cfg}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$, whose states $\mathcal{T}_{Cfg}$ are configurations, and $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$ is the transition relation defined in the previous section (Definition 2). Our goal is to have a definition of equivalence that is equally suitable for terminating programs and non-terminating ones and for symbolic and concrete ones.

A natural approach (already chosen by [1]) is to use *strong bisimulation*: a symmetrical relation $R \subseteq \mathcal{T}_{Cfg} \times \mathcal{T}_{Cfg}$ is a strong bisimulation if for all $(\gamma_1, \gamma_2) \in R$, when $\gamma_1 \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_1'$, there is a transition $\gamma_2 \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_2'$ such that $(\gamma_1', \gamma_2') \in R$. However, for our purpose such relations are too strong; e.g., the assignment `i = 2` is not equivalent to the sequence `i = 1;i = 2` because, starting from `i = 0`, the former reaches `i = 2` in one semantical step, whereas the latter cannot.

Hence, we need to alter strong bisimulation for our purposes. We do it, first, by removing the constraint that each step of one program is matched by exactly one step of the other one, and second, by requiring that our relation be bounded from above by a certain relation $O \subseteq \mathcal{T}_{Cfg} \times \mathcal{T}_{Cfg}$ called the *observation relation.*

Definition 3 (O-weak bisimulation). *An O-*weak bisimulation *is a relation $R \subseteq O$ satisfying: for all $(\gamma_1, \gamma_2) \in R$,*

- *if $\gamma_1 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma_1'$ then $\gamma_1' {\Rightarrow^*}^{\mathcal{T}}_{\mathcal{S}} \gamma_1''$ and $\gamma_2 {\Rightarrow^*}^{\mathcal{T}}_{\mathcal{S}} \gamma_2''$, for some $(\gamma_1'', \gamma_2'') \in R$*
- *if $\gamma_2 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma_2'$ then $\gamma_1 {\Rightarrow^*}^{\mathcal{T}}_{\mathcal{S}} \gamma_1''$ and $\gamma_2' {\Rightarrow^*}^{\mathcal{T}}_{\mathcal{S}} \gamma_2''$ for some $(\gamma_1'', \gamma_2'') \in R$.*

In the sequel we assume O to be an arbitrary, fixed parameter to our definitions. We omit it and only write "weak bisimulation" instead of "O-weak bisimulation". We now have our definition of program (actually, of configuration) equivalence:

Definition 4 (Configuration Equivalence). *Configurations γ_1, γ_2 are* equivalent*, written $\gamma_1 \sim \gamma_2$, if there is a weak bisimulation R such that $(\gamma_1, \gamma_2) \in R$.*

Example 3. The following configurations: $\gamma_1 \triangleq \langle\langle \texttt{x = 2}\rangle_{\mathsf{k}}\langle \texttt{x} \mapsto 0\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ and $\gamma_1' \triangleq \langle\langle \texttt{x = 1; x = x+1}\rangle_{\mathsf{k}}\langle \texttt{x} \mapsto 0\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ are equivalent when O is defined by requiring that `x` has the same value in γ_1, γ_2. The "witness" weak bisimulation R for the equivalence $\gamma_1 \sim \gamma_1'$ is defined by $\{(\gamma_1, \gamma_1'), (\gamma_2, \gamma_2)\}$, where $\gamma_2 \triangleq \langle\langle\cdot\rangle_{\mathsf{k}}\langle \texttt{x} \mapsto 2\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$.

The relation O gives us quite a lot of expressiveness for capturing various kinds of program equivalences. For example, *partial* equivalence [2] is: two programs are equivalent if, whenever presented with the same input, if they both terminate they produce the same output. This can be encoded by including cells in the configuration for the input and output, and by including in O the pairs of configurations satisfying: if their programs are both empty and their inputs are equal then their outputs are equal. Also, *full* equivalence from [2] is: two programs are equivalent if, whenever presented with the same input, they either both terminate and produce the same output, or they both do not terminate. This is captured by adding to the above relation all pairs of configurations from which there is an infinite execution starting from both configurations of the pair.

5 A Logic for Program Equivalence

We present in this section a logic for program equivalence. We first present the logic's syntax, then its semantics, and finally the notion of validity for formulas.

Definition 5 (Formulas). *A formula is an expression of the form $\pi_1 \sim \pi_2$ if C where $\pi_1, \pi_2 \in T_{\Sigma,Cfg}(Var)$ are basic patterns and $C \in T_{\Sigma,Bool}(Var)$.*

Example 4. Assume that the signature Σ for the language IMP contains a predicate `isModified` : *Id* $\times$ *Stmt* $\to$ *Bool*, expressing the fact that the value of the given identifier is modified by the semantics of the given statement. A formula expressing the equivalence of the programs in Example 1 is

$$\begin{array}{l}\langle\langle \texttt{for}\ I\ \texttt{from}\ A\ \texttt{to}\ B\ \texttt{do\{}S\texttt{\}}\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \quad \sim \\ \langle\langle I\ \texttt{=}\ A\,\texttt{;while}\ I\ \texttt{<=}\ B\ \texttt{do\{}S\,\texttt{;}\ I\ \texttt{=}\ I\texttt{+1\}}\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \\ \ \textit{if}\ \ \texttt{not isModified(}\ I, S\ \texttt{)}\end{array}$$

where M a variable of sort *Map*. The condition says that the loop counter I is not modified in the loop body S. It is essential for the formula's validity.

We now define two semantics for formulas $f \triangleq \pi_1 \sim \pi_2$ *if* C. The first one, denoted by $(\!|f|\!)$, is the set of pairs of configurations γ_1, γ_2 that satisfy, respectively, the patterns $\pi_1 \wedge C$ and $\pi_2 \wedge C$ by means of one valuation (the same valuation for both γ_1, γ_2). The second one, denoted by $[\![f]\!]$, excludes from $(\!|f|\!)$ the pairs of configurations from which at least one component eventually leads to a deadlock.

Definition 6 (Semantics).
$(\!|f|\!) \triangleq \{(\gamma_1, \gamma_2) \mid \exists \rho : \mathit{Var} \to \mathcal{T}.(\gamma_i, \rho) \models \pi_i \wedge C, i = 1, 2\}$, *and*
$[\![f]\!] \triangleq \{(\gamma_1, \gamma_2) \in (\!|f|\!) \mid \forall i \in \{1, 2\} \forall \gamma \in \mathcal{T}_{\mathit{Cfg}}.\, \gamma_i \Rightarrow^{*\mathcal{T}}_{\mathcal{S}} \gamma$ *implies* γ *is no deadlock*$\}$.

We now define what it means for a formula f to be *valid*. Intuitively, we want to capture the idea that all configurations pairs $(\gamma_1, \gamma_2) \in [\![f]\!]$ satisfy $\gamma_1 \sim \gamma_2$ according to Definition 4. We use the $[\![\cdot]\!]$ semantics (not the $(\!|\cdot|\!)$ one) because we are not interested in deadlocks. This is not really a restriction since deadlocks can be turned into final configurations by adding rules and, e.g., setting the content of some cell, say, `error`, to some value encoding the deadlock situation.

Definition 7 (Validity). *A formula f is valid, written $\mathcal{S} \models f$, if $[\![f]\!] \neq \emptyset$ whenever $(\!|f|\!) \neq \emptyset$, and for all $\gamma_1, \gamma_2 \in [\![f]\!]$, $\gamma_1 \sim \gamma_2$.*

Note that f is (vacuously) valid if $(\!|f|\!) = \emptyset$, and that f is not valid when $(\!|f|\!) \neq \emptyset$ and $[\![f]\!] = \emptyset$ because in this case all the concrete configurations in $(\!|f|\!)$ lead to deadlocks.

6 Auxiliary Operations: Derivatives and Conjunction

Our proof system consists in symbolically executing formulas according to the semantics of the language $\mathcal{L}$. This is achieved using the notion of *derivative*.

Definition 8 (Derivatives). *Given a formula $g \triangleq \pi_1 \sim \pi_2$ if C, its derivatives are the formulas in the set $\Delta(g) = \Delta^l(g) \cup \Delta^r(g)$, where $\Delta^l(g), \Delta^r(g)$ are the smallest sets defined by: for each $(l \wedge C' \Rightarrow r) \in \mathcal{S}$, $\sigma^l \in U(\pi_1, l)$, $\sigma^r \in U(\pi_2, r)$:*

– $(r\sigma^l \sim \pi_2\ \textit{if}\ (C \wedge C')\sigma^l \wedge \bigwedge\sigma^l) \in \Delta^l(g)$,
– $(\pi_1 \sim r\sigma^r\ \textit{if}\ (C \wedge C')\sigma^r \wedge \bigwedge\sigma^r) \in \Delta^r(g)$

where $\bigwedge\sigma \triangleq \bigwedge_{x\in dom(\sigma)}(x = \sigma(x))$*, and* $dom(\sigma)$ *denotes the subset of the gobal set Var of variables where the substitution* σ *is not the identity. We naturally extend derivatives to sets* F *of formulas by* $\Delta(F) = \bigcup_{f\in F}\Delta(f)$*.*

Remark 1. In Definition 8 we assume $var(l) \cap var(g) = \emptyset$, which can always be obtained by renaming the variables in the rewrite rule.

Example 5. Let B be a variable of sort *Bool* and S_1, S_2 be variables of sort *Stmt*. We consider the formula f below and compute its left-derivatives:

$$\langle\langle \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \sim \langle\langle \texttt{if } B' \texttt{ then } S_2 \texttt{ else } S_1\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \\ \textit{if } B' = \neg B$$

The rules with a nonempty set of unifiers with the patterns in the formula are

$$\langle\langle(\texttt{if } \mathit{true} \texttt{ then } S_1' \texttt{ else } S_2') \curvearrowright S\rangle_{\mathsf{k}}\langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1' \curvearrowright S\rangle_{\mathsf{k}}\langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$$
$$\langle\langle(\texttt{if } \mathit{false} \texttt{ then } S_1' \texttt{ else } S_2') \curvearrowright S\rangle_{\mathsf{k}}\langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_2' \curvearrowright S\rangle_{\mathsf{k}}\langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$$

The formula f has two left-derivatives, i.e., $\Delta^l(f)$ are the formulas in the set

$$\langle\langle S_1\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \sim \langle\langle \texttt{if}\, B' \texttt{ then } S_2 \texttt{ else } S_1\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \ \textit{if } B'{=}\neg B \wedge B{=}\mathit{true}$$
$$\langle\langle S_2\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \sim \langle\langle \texttt{if}\, B' \texttt{ then } S_2 \texttt{ else } S_1\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \ \textit{if } B'{=}\neg B \wedge B{=}\mathit{false}$$

where $B = \mathit{true}$ and $B = \mathit{false}$ are induced by the symbolic unifiers: $B \mapsto \mathit{true}$, $S_1' \mapsto S_1$, $S_2' \mapsto S_2$, $M' \mapsto M$ and, respectively, $B \mapsto \mathit{false}$, $S_1' \mapsto S_1$, $S_2' \mapsto S_2$, $M' \mapsto M$. The superfluous equalities $S_1' = S_1$, $S_2' = S_2$, $M' = M$ were removed from conditions since S_1', S_2', and M' do not occur in the rest of the formula.

Another auxiliary operation used in our proof system is *conjunction* of formulas. We need it in order to compute the subsets of configuration pairs, denoted by formulas, which are included in the observation relation O (cf. Section 4).

Definition 9. *For formulas* $f : \pi_1 \sim \pi_2$ *if* C *and* $g : \pi_1' \sim \pi_2'$ *if* C'*, let* $f \wedge g = \{\pi_1\sigma_1 {\sim} \pi_2\sigma_2 \text{ if } (C \wedge C')(\sigma_1 \cup \sigma_2) \wedge \bigwedge\sigma_1 \wedge \bigwedge\sigma_2 \mid \sigma_1 \in U(\pi_1, \pi_1'),\ \sigma_2 \in U(\pi_2, \pi_2')\}$.

Example 6. Let f be the formula in Example 5 and let g denote the formula $\langle\langle P_1\rangle_{\mathsf{k}}\langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \sim \langle\langle P_2\rangle_{\mathsf{k}}\langle M''\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ *if* $M' = M''$. We denote by π_1, π_1' and π_2, π_2' their left and right-hand sides, respectively. Then, $U(\pi_1, \pi_1')$ can be computed by matching, and consists of the unique substitution $\sigma_1 = (P_1 \mapsto \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2, M' \mapsto M)$. Similarly, $U(\pi_2, \pi_2')$ consists of the substitution $\sigma_2 = (P_2 \mapsto \texttt{if } B' \texttt{ then } S_2 \texttt{ else } S_1, M'' \mapsto M)$. Thus, if we remove the conditions $M' = M''$, $\bigwedge\sigma_1$, and $\bigwedge\sigma_2$ (which are superfluous here since they constrain variables not occuring in the rest of the result), $f \wedge g$ is syntactically equal to f. This is consistent with the fact that $\wedge$ is, semantically speaking, intersection, because we have $(\!|f|\!) \subseteq (\!|g|\!)$ and thus $(\!|f \wedge g|\!) = (\!|f|\!) \cap (\!|g|\!) = (\!|f|\!)$.

7 A Circular Proof System

In this section we define a three-rule proof system for proving program equivalence. It is inspired from *circular coinduction* [12], a coinductive proof technique for infinite data structures and coalgebras of expressions [16].

Remember that we have fixed an observation relation O. We assume a set of formulas Ω such that $(\!|\Omega|\!) = O$. We also assume that for all $h \in \Omega$ and for all formula f, the conjunction $f \wedge h$ can be computed according to Definition 9:

Assumption 3. *For all* $(\pi_1 \sim \pi_2$ *if* $C) \in \Omega$ *and all* $\pi \in T_{\Sigma,Cfg}(Var)$ *with* $(var(\pi_1) \cup var(\pi_2)) \cap var(\pi) = \emptyset$*, there are two finite, possibly empty sets* $U(\pi, \pi_1)$ *and* $U(\pi, \pi_1)$ *of symbolic unifiers of* π, π_1 *and of* π, π_2*, respectively.*

Let also $\vdash$ be an entailment relation satisfying $\mathcal{S}, F \vdash g$ implies ($\mathcal{S} \models g$ or $(\!|g|\!) \subseteq (\!|F|\!)$). The set Ω and the relation $\vdash$ are parameters of our proof system:

Definition 10 (Circular Proof System).

$$\textsf{[Axiom]}\ \frac{}{\mathcal{S}, F \vdash^{\circlearrowleft} \emptyset}$$

$$\textsf{[Reduce]}\ \frac{\mathcal{S}, F \vdash g \qquad \mathcal{S}, F \vdash^{\circlearrowleft} G}{\mathcal{S}, F \vdash^{\circlearrowleft} G \cup \{g\}}$$

$$\textsf{[Derive]}\ \frac{\mathcal{S}, F \cup F' \vdash^{\circlearrowleft} G \cup \Delta(g) \qquad \mathcal{S}, g \wedge \Omega \vdash F'}{\mathcal{S}, F \vdash^{\circlearrowleft} G \cup \{g\}}\ \textit{if}\ \Delta(g) \neq \emptyset$$

where $g \wedge \Omega$ *denotes the set* $\{g \wedge h \mid h \in \Omega\}$.

[Axiom] says that when an empty set of goals is reached, the proof is finished.

The [Reduce] rule says that if a given goal g from the current set of goals $G \cup \{g\}$ is discharged by the entailment $\vdash$ then it is eliminated from the goals.

The last rule, [Derive], is the most complex. It says that any given goal g from the current set of goals, with a nonempty set $\Delta(g)$ of derivatives, can be replaced in the goals to be proved with the set $\Delta(g)$; and, simultaneously, any set of formulas F' that can be $\vdash$-entailed from $\mathcal{S}, g \wedge \Omega$ can be added as hypotheses. Note that the application of the [Derive] rule is nondeterministic in the choice of hypotheses F', which depend on the parameters $\vdash$ and Ω of the proof system.

Theorem 1 (soundness of $\vdash^{\circlearrowleft}$). *Let* Γ *be a set of formulas such that* $(\!|\Gamma|\!) \subseteq (\!|\Omega|\!)$ *and for all* $g \in \Gamma$, $[\![g]\!] \neq \emptyset$*. If* $\mathcal{S} \vdash^{\circlearrowleft} \Gamma$ *then* $\mathcal{S} \models \Gamma$*.*

Note that we require $(\!|\Gamma|\!) \subseteq (\!|\Omega|\!)$ because otherwise the goals Γ have no chance of being valid. The asumption for all $g \in \Gamma$, $[\![g]\!] \neq \emptyset$ (that implies $(\!|g|\!) \neq \emptyset$) is made for ensuring that g is not *vacuously* valid. Note also that initially, the set of hypotheses, denoted by F in the proof system, is empty: $\mathcal{S} \vdash^{\circlearrowleft} \Gamma$ is $\mathcal{S}, \emptyset \vdash^{\circlearrowleft} \Gamma$.

We now show that the circular proof system, when it terminates, always provides an answer (positive or negative) to the question $\mathcal{S} \models \Gamma$. Thus, in addition to soundness we have a *weak completeness* result. The result is "weak" because it assumes termination of the proof system. It ensures that we have a decision procedure for the equivalence of concrete, terminating programs.

In order to achieve weak completeness we need the following adaptations of Definition 8: we only keep the formulas with a *satisfiable condition*, i.e., we eliminate "empty" formulas f with $(\!|f|\!) = \emptyset$. We also need additional assumptions. The first one says that non-derivable goals g that denote observationally equivalent configuration pairs are valid, and are discharged by the entailment $\vdash$. The second one says that deadlocks are not observationally equivalent to anything.

Assumption 4. *For all formulas g such that $(\!|g|\!) \subseteq (\!|\Omega|\!)$ and $\Delta(g) = \emptyset$, $\mathcal{S} \vdash g$; and for all configurations γ_1, γ_2, if γ_1 or γ_2 are deadlocks then $(\gamma_1, \gamma_2) \notin (\!|\Omega|\!)$.*

Theorem 2 (Weak Completeness of $\vdash^{\circlearrowleft}$). *Assume $\mathcal{S} \models \Gamma$ and the proof system $\vdash^{\circlearrowleft}$ terminates on Γ. Then, $\mathcal{S} \vdash^{\circlearrowleft} \Gamma$.*

Given a set of goals Γ, the proof system $\vdash^{\circlearrowleft}$ may *terminate successfully* on it, which means it generates a tree that has at least one "empty" leaf (generated by [Axiom]). The proof system may also *terminate unsucessfully* when it generates a finite tree and cannot expand it (i.e., it is blocked) and moreover that tree does not have any empty leaf. The proof system terminates on Γ if it terminates either sucessfully or unsucessfully. Weak completeness thus says that if a set of goals is valid and the proof system terminates on it, then it terminates successfully.

Together, the soundness and weak completeness say that, if the proof system applied to a given set of goals terminates, then termination is successful if and only if the set of goals is valid. That is, when it terminates, the proof system correctly solves the program-equivalence problem. Of course, termination cannot be guaranteed, because the equivalence problem is undecidable. It does terminate on goals in which both programs terminate (because eventually the set of derivatives becomes empty) and also for goals in which one or both of the programs does not terminate, provided they behave in a certain regular way.

Example 7. We show the application of our proof system for proving the equivalence of our *for* and *while* programs formalised as the validity of the formula f (in which we assume for simplicity that the symbolic statement S is terminating; non-terminating statements can be handled as well but complicate the example):

$$\begin{array}{l}\langle\langle \texttt{for}\ I\ \texttt{from}\ A\ \texttt{to}\ B\ \texttt{do\{}\,S\,\texttt{\}}\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \quad \sim \\ \langle\langle I\ \texttt{=}\ A\,\texttt{;while}\ I\ \texttt{<=}\ B\ \texttt{do\{}\,S\ \texttt{;}\ I\ \texttt{=}\ I\,\texttt{+1\}}\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \\ \ \mathit{if}\ \ \texttt{not isModified(}\,I, S\,\texttt{)}\end{array} \qquad (2)$$

when the observation relation is denoted by the set $\Omega = \{\langle\langle P_1\rangle_{\mathsf{k}}\langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \sim \langle\langle P_2\rangle_{\mathsf{k}}\langle M''\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\ \mathit{if}\, M' = M''\}$. The observation relation says that two configurations are observationally equivalent whenever they have equal environments.

The first applied rule is [Derive], which adds to the initially empty set of hypotheses the formula f, simulataneously replacing it in the goals with $\Delta(f)$. (f can be added to the hypotheses because $(\!|f|\!) \subseteq (\!|\Omega|\!)$, which implies $\Omega \wedge f \vdash f$).

After a certain number of applications of the [Derive] rule, the set of goals becomes (after some simplifications, which consist in removing goals with unsatisfiable conditions and logically simplifying the conditions of the remaining goals; note that A and B became (symbolic) values due to the `strict` attribute):

$$\langle\langle\rangle_{\mathsf{k}}, \langle \mathit{update}(M, I, A)\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \ \sim\ \langle\langle\rangle_{\mathsf{k}}, \langle \mathit{update}(M, I, A)\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\ \mathit{if}\, A >_{\mathit{Int}} B$$

$$\langle\langle \mathtt{for}\ I\ \mathtt{from}\ A +_{\mathit{Int}} 1\ \mathtt{to}\ B\ \mathtt{do\{}S\mathtt{\}}\rangle_{\mathsf{k}}, \langle \mathit{followup}(S, \mathit{update}(M, I, A))\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \quad \sim$$
$$\langle\langle I\,\mathtt{=}\,A +_{\mathit{Int}} 1\mathtt{;while}\,I\,\mathtt{<=}\,B\,\mathtt{do\{}S\mathtt{;}I\,\mathtt{=}\,I\,\mathtt{+}\,1\mathtt{\}}\rangle_{\mathsf{k}}, \langle \mathit{followup}(S, \mathit{update}(M, I, A))\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$$
$$\mathit{if}\ \ \mathtt{not\ isModified(}\ I, S\ \mathtt{)} \wedge A \leq_{\mathit{Int}} B$$

where $\mathit{followup}(S, M)$ denotes the effect of executing statement S on map M. Recall that S is terminating, so $\mathit{followup}(S, M)$ is defined. The fact that I is not modified by S is expressed by the equation $\mathit{followup}(S, \mathit{update}(M, I, V)) = \mathit{update}(M, I, V))$, assumed to hold in the domain of maps for all concrete instances of S that do not modify I. Moreover, for each terminating concrete instance P of S, $\mathit{followup}(P, M) = M'$ iff $\langle\langle P\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \Rightarrow^{*\mathcal{T}}_{\mathcal{S}} \langle\langle\rangle_{\mathsf{k}}, \langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$.

The first goal is discharged by the [Reduce] rule (based on the fact that the $\vdash$ relation "knows" that goals with same left and right-hade side are valid). The second goal f' is actually an *instance* of the first one: i.e., $(\!|f'|\!) \subseteq (\!|f|\!)$ since any concrete instance of f' is also a concrete instance of f. Thus, $\mathcal{S}, f \vdash f'$, and since f was added to the set hypotheses by the first application of [Derive], f' is eliminated by the [Reduce] rule. The set of goals to be proved is now empty; the proof system has terminated successfully, meaning that the formula f is valid.

8 Conclusion and Future Work

We have presented a definition for program equivalence, a logic that encodes this definition in its formulas, and a proof system for the logic, which is proved sound and weakly complete. A prototype implementation for the proof system in the $\mathbb{K}$ framework was also presented and illustrated on a simple but paradigmatic example of equivalent programs in a language IMP defined in the $\mathbb{K}$ framework.

The proposed approach is general: it does not depend on $\mathbb{K}$ and IMP but only requires a formal semantics of the language of interest as a term-rewriting system. The chosen equivalence relation is a weak bisimulation, which is parametric in a certain observation relation. We show the approach is applicable for concrete and symbolic programs, as well as for terminating and non-terminating ones.

A prototype that implements the approach for the IMP language has been developed in the $\mathbb{K}$ Framework. The implementation can be tested using the on-line interface[1] of the $\mathbb{K}$ tool.

Future Work. We are currently applying our deductive system for proving the correctness of a compiler between two languages (as part of another project we are involved in). The source language is a stack-based language with control

[1] `http://fmse.info.uaic.ro/tools/K/?tree=examples/prog-equiv/peq.k`

structures (loops, conditionals, dynamical function definitions). The target is also stack-based but only has (possibly, conditional) jumps. The correctness of the compiler amounts to proving the equivalence of several pairs of symbolic programs; in each pair, one component denotes a source-language control structure, and the other component is the translation of that control structure in the target language using jumps. We are also planning to combine our program-equivalence verification with matching logic [14] verification. Matching logic is an automatic, language-independent formal verification framework for languages with a rewrite-based semantics. The idea is to prove matching logic properties on programs in the source language, and guarantee, via the compiler's correctness that the compiled programs in the target language satisfy those properties as well.

References

1. Kundu, S., Tatlock, Z., Lerner, S.: Proving optimizations correct using parameterized program equivalence. In: Programming Languages Design and Implementation, pp. 327–337 (2009)
2. Godlin, B., Strichman, O.: Inference rules for proving the equivalence of recursive procedures. Acta Inf. 45(6), 403–439 (2008)
3. Godlin, B., Strichman, O.: Regression verification: proving the equivalence of similar programs. Software Testing, Verification and Reliability (2012), 10.1002/stvr.1472
4. Chaki, S., Gurfinkel, A., Strichman, O.: Regression verification for multi-threaded programs. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 119–135. Springer, Heidelberg (2012)
5. Leroy, X.: Formal verification of a realistic compiler. Comm. ACM 52(7), 107–115 (2009)
6. Necula, G.C.: Translation validation for an optimizing compiler. In: PLDI, pp. 83–94. ACM (2000)
7. Pitts, A.M.: Operational semantics and program equivalence. In: Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.) APPSEM 2000. LNCS, vol. 2395, pp. 378–412. Springer, Heidelberg (2002)
8. Arons, T., et al.: Formal verification of backward compatibility of microcode. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 185–198. Springer, Heidelberg (2005)
9. Craciunescu, S.: Proving the equivalence of CLP programs. In: Stuckey, P.J. (ed.) ICLP 2002. LNCS, vol. 2401, pp. 287–301. Springer, Heidelberg (2002)
10. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: Symdiff: A language-agnostic semantic diff tool for imperative programs. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 712–717. Springer, Heidelberg (2012)
11. Somenzi, F., Kuehlmann, A.: Equivalence Checking. In: Electronic Design Automation For Integrated Circuits Handbook, vol. 2, ch. 4. Taylor & Francis (2006)
12. Roşu, G., Lucanu, D.: Circular coinduction: A proof theoretical foundation. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) CALCO 2009. LNCS, vol. 5728, pp. 127–144. Springer, Heidelberg (2009)
13. Roşu, G., Şerbănuţă, T.-F.: An Overview of the K Semantic Framework. Journal of Logic and Algebraic Programming 79(6), 397–434 (2010)

14. Roşu, G., Stefanescu, A.: Checking reachability using matching logic. In: Proceedings of the 27th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2012). ACM (2012) (to appear)
15. Arusoaie, A., Lucanu, D., Rusu, V.: A Generic Approach to Symbolic Execution. Research Report RR-8189, INRIA, `http://hal.inria.fr/hal-00766220/`
16. Bonsangue, M., Caltais, G., Goriac, E.-I., Lucanu, D., Rutten, J., Silva, A.: A decision procedure for bisimilarity of generalized regular expressions. In: Davies, J. (ed.) SBMF 2010. LNCS, vol. 6527, pp. 226–241. Springer, Heidelberg (2011)

Structural Transformations for Data-Enriched Real-Time Systems*

Ernst-Rüdiger Olderog and Mani Swaminathan

Department of Computing Science
University of Oldenburg, Germany
{olderog,mani.swaminathan}@informatik.uni-oldenburg.de

Abstract. We investigate *structural transformations* for easier verification of real-time systems with shared data variables, modelled as networks of *extended timed automata* (ETA). Our contributions to this end are: (1) An operator for *layered composition* of ETA that yields *communication closed* equivalences under certain *independence* and / or *precedence* conditions. (2) Two *reachability preserving* transformations of *separation* and *flattening* for reducing (under certain cycle conditions) the number cycles of the ETA. (3) The interplay of the three structural transformations (separation, flattening, and layering), illustrated on an enhanced version of Fischer's *real-time mutual exclusion* protocol.

1 Introduction

Reasoning about networks of (real-time) systems is much easier when the execution of the system components is viewed sequentially, as opposed to corresponding distributed or concurrent representations. Transformations of distributed system representations to equivalent *layered* (i.e., sequential) representations were first explored in [1] through a notion of *communication closedness* between system components. Such a layered transformation was subsequently investigated in [2] for a process algebra based on hierarchical graphs, with an operator for *layered composition* (intermediate between sequential and parallel composition) that formalized *equivalences* between the distributed and layered system representations through the *Communication Closed Layer* (CCL) laws, by exploiting *independence* between system components. Real-time extensions to this process algebra were presented in [3], where CCL laws were shown to hold under certain *timing conditions*, even in the absence of cross-component independence.

The layered transformation used in the *assertion-based* reasoning techniques of the above works was recently adapted in [4] for *automatic verification* of real-time systems modelled as *timed automata* (TA) [5]. Our layered transformation in [4] aimed for state-space reduction in TA networks, based on *transition independence* as studied for *partial order reduction* of TA [6,7,8,9]. We enhance here

* This work has been partially funded by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, www.avacs.org).

the layered transformation in [4] for TA, and complement this by the transformation techniques of *separation* and *flattening.* The structure of this paper and the enhancements therein w.r.t [4] are as follows :

(1) We considered in [4] networks of TA under *local time semantics*, as in many works on partial order reduction for TA (cf. [6,7,8,9]), where the clocks of each constituent TA evolve independently so as to induce greater timing-based independence, but at the expense of extra *reference clocks* for resynchronization (cf. [6]). In this paper, we instead work with networks of TA extended with shared data variables (termed extended timed automata (ETA)), having *synchronous clocks* across the constituent ETA, as supported by the well-established ETA model-checker UPPAAL [10]. Dependencies in such ETA networks arise due to (a) the read-write interference of the variables shared across the ETA (b) the global timing constraints induced by synchronous clocks.

Section 2 of this paper reviews ETA and their compositional constructs, and establishes communication closed equivalences that exploit the absence of dependencies due to (a) and (b) above. Notions of non-interference are introduced for dealing with (a), while (b) is handled by *wrapped* ETA that mimic local time semantics in a network even with synchronous clocks.

Section 3 of this paper establishes communication closed equivalences for ETA with synchronous clocks that exploit global-time-induced *precedence relations*, in the presence of shared variable and clock dependencies between ETA.

(2) The explicit passage of control (from one sequential phase of the system to the next) necessary for applying (non-interference- or precedence-based) layered transformations may however not be directly apparent from the system's structure, owing to multiple nested cycles that often arise while modelling reactive distributed real-time systems as ETA networks. We therefore introduce in Section 4 the transformations of *separation* and *flattening* as (reachability preserving) pre-processing steps that (under certain cycle conditions on the ETA) reduce the nesting depth and the number of cycles in ETA networks. Communication closedness (via appropriate non-interference and/or precedence conditions) may then be easily investigated on such separated and flattened ETA, such that the verification of reachability properties may be rendered almost trivial.

(3) The interplay of the three structural transformations (separation, flattening, and layering) is illustrated in Section 5 on an enhanced version of Fischer's real-time mutual exclusion protocol for two critical sections.

Section 6 concludes the paper with further comparisons to related work. A full version of this paper is available at [11], whose Appendix A details the syntax and semantics of ETA and their compositional constructs, whose Appendix B includes proofs of the results stated in this paper, and whose Appendix C discusses the generality of these results.

2 ETA, Compositions, and CCL

We briefly review the *Extended Timed Automata* (ETA) model for (networks of) real-time systems enriched with *shared data variables.* Various compositional operators for ETA are also introduced, along with the notion of transition independence yielding communication closed equivalences for suitably wrapped ETA in the presence of synchronous clocks.

Extended Timed Automata. Let $(\alpha, \beta, \cdots \in)\ \Sigma$ be a finite *alphabet* of *channels*. For each channel $\alpha \in \Sigma$ there are two *actions*: $\alpha?$ denotes *input* on α and $\alpha!$ *output* on α, where $\alpha?, \alpha! \notin \Sigma$. We consider two different *internal* actions $\tau, \varepsilon \notin \Sigma$, where τ results only from *synchronization*. The set of all actions over Σ is denoted by $(a, b, \cdots \in)\ \Sigma_{?!} = \{\alpha? \mid \alpha \in \Sigma\} \cup \{\alpha! \mid \alpha \in \Sigma\} \cup \{\tau, \varepsilon\}$. In the context of parallel composition, input and output are *complementary* actions that can synchronize yielding τ. For an action $a \in \Sigma_{?!} \setminus \{\tau, \varepsilon\}$, its complementary action is denoted by $\overline{a}$, i.e., $\overline{\alpha?} = \alpha!$, and vice-versa.

Let $(v \in)\ V$ be a finite set of *data variables* ranging over a finite value set D. The set $(\psi_D \in)\ \Psi(V)$ of *data expressions* over V is the set of expressions involving variables of V and the usual arithmetic operators $+, -, \cdots$. The set $(\phi_D \in)\ \Phi(V)$ of *data constraints* over V is the set of Boolean constraints over variables in V involving the usual arithmetic $(+, -, \cdots)$ and relational $(<, \leq, >, \geq)$ operators. A *data valuation* assigns to each data variable in V a value in D. If $|V| = m$, a data valuation is identified with a point in D^m, denoted typically by $\vec{u}, \vec{v}$ etc.

Let $(x, y, \cdots \in)\ C$ be a finite set of *clocks*. The set $(\phi \in)\ \Phi(C)$ of *clock constraints* over C has the following syntax: $\phi ::= x \bowtie k \mid \phi_1 \wedge\ \phi_2$, where $x \in C$, $k \in \mathbb{N}$, and $\bowtie\ \in \{<, \leq, \geq, >\}$. The set $(g \in)\ G(C, V)$ of *guards* has the syntax: $g ::= \phi \mid \phi_D \mid g_1 \wedge g_2$, where $\phi \in \Phi(C)$ and $\phi_D \in \Phi(V)$.

A *clock valuation* assigns a non-negative real value to each clock in C. If $|C| = n$, a clock valuation is identified with a point in $\mathbb{R}^n_{\geq 0}$, which we typically denote by $\vec{x}, \vec{y}$ etc. By $\vec{0}$ we denote the clock valuation where all clocks are set to 0, while $\vec{v_0} \in D^m$ denotes a designated initial data valuation. A *reset* operation is an assignment $x := 0$ to a clock $x \in C$, or an assignment $v := \psi_D$ to a data-variable $v \in V$, involving a data expression $\psi_D \in \Psi(V)$.

We may then define an *extended timed automaton* (ETA) A as a tuple $A = (L, \Sigma, C, V, l_0, l_F, Inv, E)$ over a set of (finitely many) locations, channel names, clocks, data variables, initial and final locations, invariants, and edges. An edge $e \in E$ is of the form $e = (l, a, g, r, l')$ with $l, l' \in L, a \in \Sigma_{?!}, g \in G(C, V)$, and r a list of reset operations. Define $target(e) = l'$ as the target location of the edge e, $act(e) = a$ as the *action of* e and $edges_A(a) = \{e \in E \mid act(e) = a\}$ as the set of edges in A with action a. For a pair of clock valuations $\vec{x}$ and $\vec{y}$ and a constant $k \in \mathbb{N}$, we denote by $\vec{x} \approx_k \vec{y}$ the *k-region-equivalence* between $\vec{x}$ and $\vec{y}$. The corresponding equivalence class for a clock valuation $\vec{x}$ is denoted $[\vec{x}]_k$, cf. Definitions 11 and 12 in Appendix A of [11].

The semantics of an ETA is given in terms of its *timed transition system*, which consists of an infinite set of *states* of the form $(l, \vec{x}, \vec{v})$, where $l \in L$, $\vec{x} \in \mathbb{R}^n_{\geq 0}$, and $\vec{v} \in D^m$. The transitions between such states result in the formation of *paths* through the timed transition system. Such a *path* π is a (possibly infinite) sequence $\pi = \langle (l_0, \vec{0}, \vec{v_0}) \overset{d_0}{\to} (l_1, \vec{x_1}, \vec{v_1}) \overset{e_1}{\to} (l_2, \vec{x_2}, \vec{v_2}) \overset{d_2}{\to} (l_3, \vec{x_3}, \vec{v_3}) \overset{e_3}{\to} \ldots \rangle$ of states with delays $d_i \in \mathbb{R}_{\geq 0}$ and edges $e_i \in E$, subject to the *initiation* and *consecution* conditions (cf. Definition 13 in Appendix A of [11]). This path π induces a *timed trace* $\xi = \langle (t_0, e_1), (t_2, e_3), \ldots \rangle$ with $t_0 = d_0$ and $\forall i \in \mathbb{N}$: $t_{2i+2} - t_{2i} = d_{2i+2}$, where ξ consists of pairs (t, e) indicating the *absolute* time instant $t \geq 0$ at which the edge $e \in E$ is executed by ETA A along path π. Note that all timestamps in π and ξ have even subscripts. The reachable state space of the ETA A, denoted $\mathit{Reach}(A)$, is then given by the set of states reachable from the initial state through transitions of all paths, with $\mathit{Reach}_i(A)$ denoting the set of reachable states of A after i iterations of its transition relation (cf. Definition 14 in Appendix A of [11]). This leads to the notion of *reachability equivalence* denoted by $\equiv$. Given two ETA A_1 and A_2, we define $A_1 \equiv A_2$ iff $\forall i \in \mathbb{N}$: $\mathit{Reach}_i(A_1) = \mathit{Reach}_i(A_2)$. Thus $\equiv$ requires equal sets of reachable states after every iteration of the transition relation.

ETA Compositions. So far we considered ETA operating in isolation. In practice, real-time systems *communicate* with each other and their environment. This results in *composite* systems with communicating components. The communication is via synchronizing actions drawn from a shared alphabet and via shared data variables. We now consider four operators for constructing composite systems: *sequential*, *step*, *parallel*, and *layered* composition (where the latter is new for (extended) timed automata), defined for ETA $A_i = (L_i, \Sigma_i, C_i, V_i, l_{0i}, l_{Fi}, \mathit{Inv}_i, E_i)$, $i = 1, 2$, with disjoint locations: $L_1 \cap L_2 = \emptyset$.

For modeling that the execution of A_1 is followed by that of A_2, it is convenient to have two composition operators at hand. *Sequential composition* $A_1; A_2$ *amalgamates* the final location l_{F1} of A_1 with the initial location l_{02} of A_2, the *step composition* $A_1 \rhd A_2$ links l_{F1} and l_{02} by an explicit *step transition* t. Formally, $A_1; A_2 = (L_1 \cup L_2 \cup \{\widetilde{l_{F1}}\}, \Sigma_1 \cup \Sigma_2, C_1 \cup C_2, V_1 \cup V_2, l_{01}, l_{F2}, \mathit{Inv}, E)$, where $\widetilde{l_{F1}}$ is a new location obtained by amalgamating l_{F1} with l_{02}. We require that there is no outgoing edge from l_{F1}, as it would otherwise be possible to reenter A_1 from A_2 via incoming edges to l_{02} and outgoing edges from l_{F1}. The set of edges E is obtained by appropriately assigning $\widetilde{l_{F1}}$ as the target or source location for edges in E_1 entering l_{F1} and for edges in E_2 entering or leaving l_{02}, cf. Definition 15 in Appendix A of [11]. In the *step composition* $A_1 \rhd A_2$ defined below, the stepping transition t allows for l_{F1} to have outgoing transitions, as no location of A_1 will be re-entered once t has been executed.

Definition 1 (Step Composition). *The* step composition $A_1 \rhd A_2$*, read* A_1 step A_2*, is defined by* $A_1 \rhd A_2 = (L, \Sigma_1 \cup \Sigma_2, C_1 \cup C_2, V_1 \cup V_2, l_{01}, l_{F2}, \mathit{Inv}, E)$*, where* $L = L_1 \cup L_2$*, with* $\mathit{Inv}(l_i) = \mathit{Inv}_i(l_i)$ *for* $l_i \in L_i$ *and* $i = 1, 2$*, and* $E = E_1 \cup E_2 \cup \{t\}$*, where* $t = (l_{F1}, \varepsilon, \mathit{true}, \emptyset, l_{02})$ *steps from* l_{F1} *to* l_{02}*.*

Alternative definitions of sequential and step compositions for timed automata may be found in [12,13]. Parallel composition $\|$ of ETA is in the CCS-style [14], i.e., parallel ETA *synchronize* on common actions but also act autonomously on all actions – the latter is modelled by *interleaving*. In order to avoid any read-write and write-write conflicts w.r.t the shared variables in the parallel ETA, we require that edges with synchronizing actions are *non-interfering*, as defined below. For an edge $e = (l, a, g, r, l')$ of an ETA A its *write-set* $\mathit{wr}(e)$ is the set of all clocks and data variables appearing on the left-hand side of one of the reset operations in r, while its *read-set* $rd(e)$ is the set of all clocks and data variables appearing in the guard g or on the right-hand side of a reset operation in r.

Definition 2 (Non-interfering edges). *Let* E_1 *and* E_2 *be sets of edges. The* non-interference relation $\not\curvearrowright \subseteq E_1 \times E_2$ *is defined for* $e_1 \in E_1$ *and* $e_2 \in E_2$ *by:* $e_1 \not\curvearrowright e_2$ *if* $rd(e_1) \cap wr(e_2) = wr(e_1) \cap rd(e_2) = wr(e_1) \cap wr(e_2) = \emptyset$.

The relation $\not\curvearrowright$ is canonically lifted to sets of edges (and consequently to ETA): $E_1 \not\curvearrowright E_2$ iff for all $e_1 \in E_1$ and $e_2 \in E_2$ we have $e_1 \not\curvearrowright e_2$. For two ETA A_1 and A_2 with respective edge-sets E_1 and E_2, we have that $A_1 \not\curvearrowright A_2$ when (1) $E_1 \not\curvearrowright E_2$, i.e., their edge-sets are non-interfering, and (2) $C_1 \cap C_2 = \emptyset$, i.e., their clock-sets are disjoint so as to eliminate timing-induced dependencies between A_1 and A_2 by the *wrapping* construction (cf. Definition 4). In the context of parallel composition $\|$, we require a more relaxed notion of *synchronized non-interference* on the constituent ETA A_1 and A_2 for $A_1 \| A_2$ to be well-formed.

Definition 3 (Synchronized non-interfering ETA). *ETA* A_1 *and* A_2 *over alphabets* Σ_1 *and* Σ_2*, respectively, are* synchronized non-interfering*, denoted* $A_1 \not\curvearrowright_{sync} A_2$*, if* $\forall a \in \Sigma_{1?!} \setminus \{\tau, \varepsilon\} : \overline{a} \in \Sigma_{2?!} \implies edges_{A_1}(a) \not\curvearrowright edges_{A_2}(\overline{a})$.

The relation $\not\curvearrowright_{sync}$ on ETA is only w.r.t synchronizing actions on common channels, and thus (unlike the more restrictive $\not\curvearrowright$ relation on ETA) does not preclude shared-variable and clock dependencies between actions on disjoint channels.

The *parallel composition* of two A_1 and A_2, with $A_1 \not\curvearrowright_{sync} A_2$, is defined by $A_1 \| A_2 = (L_1 \times L_2, \Sigma_1 \cup \Sigma_2, C_1 \cup C_2, V_1 \cup V_2, (l_{01}, l_{02}), (l_{F1}, l_{F2}), Inv, E)$, where $\forall (l_1, l_2) \in L_1 \times L_2 : Inv(l_1, l_2) = Inv_1(l_1) \wedge Inv_2(l_2)$, and E is constructed according to a CCS-style synchronization and interleaving, cf. Definition 16 of Appendix A of [11]. As mentioned in the introduction, a real-time distributed system often consists of (sequential) phases that execute in parallel on multiple platforms, wherein a transition (edge) within a given phase can execute only after all *dependent* transitions (edges) in each preceding phase have been executed. It is clear that the non-interference relation of Definition 2 is sufficient to ensure independence in the untimed setting, where dependencies are induced only by shared variables. In the timed setting of ETA, however, the clocks of the various system components evolve *synchronously*, resulting in *timing-induced* dependencies even in the presence of disjoint sets of clocks. In contrast to several works on the partial order reduction of TA (e.g., [6,7,8]) that deal with such timing-induced dependencies by imposing the semantic condition of *local time* (where, in addition to mutual disjointness, the clocks of the constituent components run entirely independent of each other), we retain here the synchrony between the clocks of the various components as in the UPPAAL model-checker, but eliminate timing-induced dependencies by *wrapping* the ETA with an initial location that admits idling for arbitrarily long periods before proceeding to its actual execution. The wrapping concept is defined as follows:

Definition 4 (Wrapped ETA). *An ETA* $A = (L, \Sigma, C, V, l_0, l_F, Inv, E)$ *is* wrapped *if* $Inv(l_0) = true$ *and every edge* $e \in E$ *leaving* l_0 *is of the form*

$e = (l_0, \varepsilon, true, r, l)$, where r resets all clocks in C to 0 and all data variables in V to their initial value $\vec{v_0}$. If A is wrapped, we denote this by writing $[A]$.

The arbitrary idling permitted in l_0 mimics local time semantics when $[A]$ is considered in the context of a (parallel) composition. Intuitively, $[A]$ is protected against time influences from components working in parallel.

We now introduce an asymmetric layered composition operator $\bullet$ (intermediate between parallel and sequential composition) that involves the non-interference relation on edges of ETA. The *layered composition* of A_1 and A_2 is given by $A_1 \bullet A_2 = (L_1 \times L_2, \Sigma_1 \cup \Sigma_2, C_1 \cup C_2, V_1 \cup V_2, (l_{01}, l_{02}), (l_{F1}, l_{F2}), Inv, E)$, where $A_1 \not\leftrightarrow_{sync} A_2$, and Inv is as in the parallel composition $A_1 \| A_2$, while E is a subset of the set of edges of $A_1 \| A_2$, as an edge of A_2 is allowed to execute in $A_1 \bullet A_2$ only after all *dependent* edges of A_1 have been executed, cf. Def. 17 and Fig. 5 in Appendix A of [11]. Theorem 1 states the CCL laws of [2] for ETA.

Theorem 1 (CCL laws for ETA). *For all ETA A_1, A_2 and B_1, B_2 with $A_1 \not\leftrightarrow B_2$ and $B_1 \not\leftrightarrow A_2$ the* communication closed layer *(CCL) laws hold:*

1. $A_1 \bullet B_2 \;=\; A_1 \| B_2$ *(Indep)*
2. $(A_1 \bullet A_2) \| B_2 \;=\; A_1 \bullet (A_2 \| B_2)$ *(CCL-L)*
3. $(A_1 \bullet A_2) \| B_1 \;=\; (A_1 \| B_1) \bullet A_2$ *(CCL-R)*
4. $(A_1 \bullet A_2) \| (B_1 \bullet B_2) \;=\; (A_1 \| B_1) \bullet (A_2 \| B_2)$ *(CCL)*

PO-Equivalence. We now formalize *partial order* (po) equivalence as a means of relating step and layered compositions. For this relationship, we have to address the fact that in a layered composition $A_1 \bullet A_2$ there may be τ-edges arising from synchronization of complementary actions, whereas such τ-edges do not arise in the step composition $A_1 \rhd A_2$. For this purpose, we introduce for a path π of $A_1 \bullet A_2$ the operation *split*(π) that *splits* every synchronization edge of π (labelled with τ) into a sequence of its constituent input and output edges, which is possible owing to the synchronized non-interference assumed for edges labelled with complementary actions (see Definition 3). Note that this non-interference implies that the order in which synchronization edges are split into constituent input and output edges is irrelevant.

Consider a finite τ-labelled path π of $A_1 \bullet A_2$ with fragments π' and π'' of the form $\pi = \pi' \xrightarrow{d_0} (l_1, \vec{x}_1, \vec{u}_1) \xrightarrow{\tau} (l_2, \vec{x}_2, \vec{u}_2) \xrightarrow{d_2} \pi''$. We then have *split*$(\pi) = \pi' \xrightarrow{d_0'} (l_1', \vec{x}_1', \vec{u}_1') \xrightarrow{a} (l_2', \vec{x}_2', \vec{u}_2') \xrightarrow{d_2'} (l_3', \vec{x}_3', \vec{u}_3') \xrightarrow{\overline{a}} (l_4', \vec{x}_4', \vec{u}_4') \xrightarrow{d_4'} \pi''$, with $a \in \Sigma_i$ and $\overline{a} \in \Sigma_{3-i}$, $i \in \{1, 2\}$, where $l_1 = l_1'$, $\vec{x}_1 \approx_k \vec{x}_1'$, $\vec{u}_1 = \vec{u}_1'$, $l_2 = l_4'$, $\vec{x}_2 \approx_k \vec{x}_4'$, $\vec{u}_2 = \vec{u}_4'$, and where $d_0, d_0', d_2, d_2', d_4' \in \mathbb{R}_{\geq 0}$, $d_0 = d_0'$, $d_2' = 0$, $d_2 = d_4'$, and k is the maximum of the clock constants appearing in A_1 and A_2. Following such a splitting of τ-edges, we may now define po-equivalence on paths of ETA.

Definition 5 (po equivalence of paths). *Let A_1 and A_2 be two ETA sharing an alphabet Σ, with Π_1 and Π_2 denoting the corresponding sets of finite paths. Let $\approx$ be a relation between the locations of A_1 and A_2. A path $\pi_1 \in \Pi_1$ is* po equivalent *to $\pi_2 \in \Pi_2$, denoted $\pi_1 \equiv_{po} \pi_2$, relative to $\approx$ on the corresponding locations, $\approx_k$ on the corresponding clock-valuations (where k is the maximum* *constant of A_1 and A_2), and identity on the corresponding data valuations, if split(π_2) can time-abstractedly be obtained from split(π_1) by repeated permutation of adjacent independent edges separated by only one time-passage.*

For $\pi_1 = \langle (l_{01}, \vec{0}, \vec{v_0}) \xrightarrow{d_0} (l_1, \vec{x}_1, \vec{u}_1) \xrightarrow{e} (l_2, \vec{x}_2, \vec{u}_2) \xrightarrow{d_2} (l_3, \vec{x}_3, \vec{u}_3) \xrightarrow{f} (l_4, \vec{x}_4, \vec{u}_4)\rangle$ and $\pi_2 = \langle (l_{02}, \vec{0}, \vec{v_0}) \xrightarrow{d'_0} (l'_1, \vec{y}_1, \vec{v}_1) \xrightarrow{f} (l'_2, \vec{y}_2, \vec{v}_2) \xrightarrow{d'_2} (l'_3, \vec{y}_3, \vec{v}_3) \xrightarrow{e} (l'_4, \vec{y}_4, \vec{v}_4)\rangle$, where $d_0, d_2, d'_0, d'_2 \in \mathbb{R}_{\geq 0}$, we say that $\pi_1 \equiv_{po} \pi_2$ relative to $\approx$ iff $l_{01} \approx l_{02}$, and $\forall 1 \leq i \leq 4 : l_i \approx l'_i$, $\vec{x}_4 \approx_k \vec{y}_4, \vec{u}_4 = \vec{v}_4$, and $e \not\frown f$. Thus, two po equivalent paths π_1 and π_2 (relative to $\approx$ on their locations, region-equivalence on their clock valuations, and identity on their data valuations) differ only in the (permutative) ordering of *independent* transitions. This definition has been adapted for ETA from [15]. The notion of po equivalence is then lifted to ETA as follows: For ETA A_1 and A_2 sharing a common alphabet Σ, with Π_1 and Π_2 denoting the corresponding sets of finite paths ending in their respective final states, we write $A_1 \equiv_{po} A_2$ iff $\forall \pi_i \in \Pi_i \ \exists \pi_{3-i} \in \Pi_{3-i} : \pi_i \equiv_{po} \pi_{3-i}$, where $i \in \{1, 2\}$.

Definition 6 (Layered Normal Form). *A (finite) path π of $A_1 \bullet A_2$ is in* layered normal form (LNF) *if it consists of consecutive edges from E_1 passing through l_{F1}, followed by consecutive edges from E_2 ending in l_{F2}.*

In a path π of $A_1 \bullet A_2$ in LNF, the A_2-transitions are *delayed* until all A_1-transitions have occurred. This may be too late because the clock constraints of the A_2-transitions may not be satisfied any more. To avoid this issue, we wrap A_2 so that starting $[A_2]$ resets all clocks of A_2. This way, we mimic a local time semantics for A_2. Now Lemma 1 states that every path of $A_1 \bullet [A_2]$ can be rewritten into a po equivalent path in LNF, which leads to Theorem 2 establishing po equivalence between layered and step compositions of ETA.

Lemma 1. *Consider an ETA A_1 that can terminate in its final location l_{F1}, and a wrapped ETA $[A_2]$. Let Π denote the set of all finite paths of $A_1 \bullet [A_2]$, and $\Pi_L \subseteq \Pi$ the subset of the paths in LNF. Then $\forall \pi \in \Pi \ \exists \pi' \in \Pi_L : \pi \equiv_{po} \pi'$.*

Theorem 2 (po equivalence between $\bullet$ and $\rhd$). *For an ETA A_1 that can terminate, and a wrapped ETA $[A_2]$, we have that $A_1 \bullet [A_2] \equiv_{po} A_1 \rhd [A_2]$.*

Note that $A_1 \bullet [A_2] \equiv_{po} A_1 \rhd [A_2]$ implies that local reachability of locations is preserved, as $\mathit{Reachloc}(A_1) \cup \mathit{Reachloc}([A_2]) = \mathit{Reachloc}(A_1 \rhd [A_2])$, where $\mathit{Reachloc}(A)$ denotes the set of reachable locations of the ETA A. Theorems 1 and 2 lead to the following corollary.

Corollary 1 (CCL laws with step). *For all ETA A_1, B_1 and all wrapped ETA $[A_2], [B_2]$ such that $A_1 \not\frown B_2$ and $B_1 \not\frown A_2$, and such that when A_1 can terminate, then so can B_1, and vice-versa, with $P = (A_1 \rhd [A_2]) \parallel (B_1 \rhd [B_2])$ and $S = (A_1 \parallel B_1) \rhd ([A_2] \parallel [B_2])$, we then have that $P \equiv_L S$.*

$\equiv_L$ between P and S here is the *layered reachability equivalence* satisfying: $\mathit{Reachloc}(S) \subseteq \mathit{Reachloc}(P)$, and for any $(l_a, l_b) \in \mathit{Reachloc}(P)$

– if $l_a \in L_{A_i}, l_b \in L_{B_i}$, then $(l_a, l_b) \in \mathit{Reachloc}(S)$,

– if $l_a \in L_{A_i}, l_b \in L_{B_{3-i}}$, then $\exists\, l'_a \in L_{A_{3-i}}, l'_b \in L_{B_i} : (l_a, l'_b) \in \mathit{Reachloc}(S) \ \wedge$ $(l'_a, l_b) \in \mathit{Reachloc}(S)$,

where $i \in \{1, 2\}$.

3 Time Precedence and Timed CCL

The non-interference conditions used in the CCL laws of the preceding section may be *syntactically* inferred from the structure of the ETA. We now introduce a semantic condition termed *time precedence* and demonstrate its use in establishing equivalences analogous to Theorem 1 and Corollary 1 for ETA networks that do not respect the non-interference conditions discussed in the previous section. This time precedence relation is defined below for ETA.

Definition 7 (Time Precedence in ETA). *For ETA A_1 and A_2 we say that A_1* precedes *A_2, denoted $A_1 \prec A_2$ if $A_1 \| A_2 \equiv A_1 ; A_2$.*

Though $\prec$ is a semantic condition, it may be easily verified by inspecting the ETA guards, as will be shown in Section 5. For the specific case of acyclic ETA, where there is no location that is syntactically reachable from itself, we may refine Definition 7 to relate time-stamps of individual edges as follows:

Definition 8 (Time Precedence of Edges). *Consider two acyclic ETA A_1 and A_2 with corresponding edge sets E_1 and E_2, and the set $\Xi(A_1 \| A_2)$ of timed traces induced by paths of $A_1 \| A_2$. Then an edge $e_1 \in E_1$ is said to* precede *an edge $e_2 \in E_2$, denoted $e_1 \prec e_2$, if*

$$\forall \xi \in \Xi(A_1 \| A_2) \ \begin{pmatrix} \exists t_2 \in \mathbb{R}_{\geq 0} : (t_2, e_2) \in \xi \\ \Rightarrow \ \ \exists t_1 \in \mathbb{R}_{\geq 0} : [(t_1, e_1) \in \xi \ \ \wedge \ \ t_1 < t_2] \end{pmatrix}.$$

The following Lemma then follows as an immediate consequence:

Lemma 2 (Edge Precedence in acyclic ETA). *Given two acyclic ETA A_1 and A_2, we have that $A_1 \prec A_2 \iff \forall e_1 \in E_1 \ \forall e_2 \in E_2 : e_1 \prec e_2$.*

The CCL laws discussed previously may then be reformulated as follows, using this semantic notion of time-precedence as an alternative side-condition in the presence of syntactic dependencies.

Theorem 3 (Timed CCL laws for ETA). *For ETA A_1, A_2 and B_1, B_2 with $A_1 \prec B_2$ and $B_1 \prec A_2$ the following* timed communication closed layer *(Timed CCL) laws hold for the reachability equivalence $\equiv$:*

1. $A_1 \bullet B_2 \ \equiv \ A_1 \| B_2$ *(Timed Indep)*
2. $(A_1 \bullet A_2) \| B_2 \ \equiv \ A_1 \bullet (A_2 \| B_2)$ *(Timed CCL-L)*
3. $(A_1 \bullet A_2) \| B_1 \ \equiv \ (A_1 \| B_1) \bullet A_2$ *(Timed CCL-R)*
4. $(A_1 \bullet A_2) \| (B_1 \bullet B_2) \ \equiv \ (A_1 \| B_1) \bullet (A_2 \| B_2)$ *(Timed CCL)*

Theorem 3 then implies the reachability equivalence $\equiv$ when $\bullet$ is replaced by ; within the expressions of Theorem 3, as stated in the following corollary:

Corollary 2. *Replacing $\bullet$ by ; within the expressions appearing in Theorem 3 yields the reachability equivalence $\equiv$.*

Unlike the CCL laws in Theorem 1, the Timed CCL laws of Theorem 3 do not hold for the equality $=$. Note also that Corollary 2 does not require the ETA to the right of the $\bullet$ to be wrapped, in contrast to Corollary 1. As $\equiv$ is not a congruence w.r.t. parallel composition, the Timed CCL laws do not yield equivalences in an arbitrary parallel context.

4 Separation and Flattening

We consider in this section two transformations on cycles in ETA. *Separation* reduces the nesting of cycles, while *flattening* reduces the number of cycles. These transformations are sound (in the sense of reachability preservation) under the assumption that the ETA involved have *memoryless cycles.* Such an assumption is justified for protocols where each cycle performs some service, and there is no need to carry over some information from one service cycle to the next. Separation was studied in [16] in the abstract setting of Kleene algebras, where, under certain conditions, a nondeterministic iteration of the form $(a + b)^*$ could be separated into a sequence a^*b^* of iterations, with a and b being regular expressions for programs in a Kleene algebra. This is in essence what we will prove for ETA in the Separation Theorem of this section. The role of the nondeterministic choice $+$ on regular expressions is played by a *union* operator $\cup$ on ETA [12].

Definition 9 (Union). *Consider ETA $A_i = (L_i, \Sigma_i, C_i, V_i, l_0, l_0, Inv_i, E_i)$, $i \in \{1, 2\}$, with an identical initial and final location l_0 such that $L_1 \cap L_2 = \{l_0\}$. $A_1 \cup A_2 = (L_1 \cup L_2, \Sigma_1 \cup \Sigma_2, C_1 \cup C_2, V_1 \cup V_2, l_0, l_0, Inv_1 \cup Inv_2, E_1 \cup E_2)$ is then the* union *of A_1 and A_2.*

Whereas in the union $A_1 \cup A_2$ possible cycles of A_1 and A_2 are glued together in their initial location, in $A_1 \rhd A_2$ the new transition t separates the A_1 cycles from the A_2 cycles so that all A_1 cycles are performed before the A_2 cycles. In contrast to the other composition operators of the previous sections requiring the location disjointness condition $L_1 \cap L_2 = \emptyset$, the operator $\cup$ in this section instead requires that A_1 and A_2 have a common *memoryless* initial location, where the notion of a location being memoryless is as defined below:

Definition 10 (Memoryless locations in ETA). *A location l of an ETA A is said to be* memoryless *if l is always entered with the initial valuations of the clocks and the data variables.*

A sufficient syntactic condition for a location l to be memoryless is that all cycles through l have strong resets. A cycle of an ETA A through a location l is said to have *strong resets* if every transition entering l resets all clocks and all data variables. To simplify the reasoning about cycles, we wish to transform the union of ETA with memoryless cycles into their step composition. The Separation Theorem shows that this transformation respects a weak reachability equivalence $\equiv_r$ sufficient for the preservation of safety properties, as seen next.

Definition 11 (Weak Reachability Equivalence). *For ETA A and A', with $A = (L, \Sigma, C, V, l_0, l_F, Inv, E)$ and $A' = (L', \Sigma', C', V', {l_0}', {l_F}', Inv', E')$, with $|C| = |C'| = n$, and $|V| = |V'| = m$, we define the* weak reachability equivalence *between A and A', denoted by $A \equiv_r A'$, (relative to a relation $\approx \subseteq L \times L'$) if $\forall\, l \in L\, \forall\, l' \in L'\, \forall \vec{x} \in \mathbb{R}^n_{\geq 0}\; \forall \vec{v} \in D^m$:*

1. *$(l, \vec{x}, \vec{v}) \in Reach(A) \;\;\Rightarrow\;\; \exists\, l' \in L' : l \approx l' \wedge (l', \vec{x}, \vec{v}) \in Reach(A')$.*
2. *$(l', \vec{x}, \vec{v}) \in Reach(A') \;\;\Rightarrow\;\; \exists\, l \in L : l \approx l' \wedge (l, \vec{x}, \vec{v}) \in Reach(A)$.*

Theorem 4 (Separation). *Consider ETA A_1 and A_2 as in Definition 9 with memoryless initial locations ${l_0}_1 = {l_0}_2$. Then $A_1 \cup A_2 \equiv_r A_1 \rhd A_2$.*

In general, $\equiv_r$ is not a congruence w.r.t. parallel composition. Nevertheless, the following theorem states its preservation by parallel instances of separation under the non-interference conditions similar to those of the CCL laws.

Theorem 5 (Separation in parallel context). *For ETA A_1, B_1, A_2, B_2 with memoryless initial locations, with A_2 and B_2 wrapped and satisfying $A_1 \not\curvearrowright B_2$ and $B_1 \not\curvearrowright A_2$, it holds that $(A_1 \cup [A_2]) \| (B_1 \cup [B_2]) \equiv_r (A_1 \rhd [A_2]) \| (B_1 \rhd [B_2])$.*

The next theorem states that an ETA with a memoryless location l can be *flattened* into one that contains fewer cycles through l, while preserving $\equiv_r$.

Theorem 6 (Flattening). *Consider an ETA $A^* = (L, \Sigma, C, V, l_0, l_F, Inv, E^*)$ with a memoryless location $l \in L$. Then $A_l = (L, \Sigma, C, V, l_0, l_F, Inv, E_l)$, where $E_l = E^* \backslash \{e \mid target(e) = l$ and there is no syntactic path from l_0 to l in E^* such that e is the first edge with $target(e) = l$ on this path$\}$, satisfies $A^* \equiv_r A_l$.*

If $E_l \subset E^*$ then A_l is a *flattened* version of A^* with a reduced number of cycles through l. Note that flattening at l_0 results in A_{l_0} being cycle-free at l_0. Next, we consider flattening of A^* in the context of a parallel composition $A^* \| B$ and state sufficient conditions for the preservation of *location reachability.*

Theorem 7 (Flattening in parallel context). *Suppose that for A^* and B, where A^* is memoryless at $l_a \in L_A$, the following holds within $A^* \| B$:*

1. *Every location of A^* is reachable from its initial location l_{0A} without visiting l_a more than once, while B stays in its initial location l_{0B}.*
2. *Every location of B is reachable from l_{0B}, while A^* stays in l_a.*
3. *If a transition entering l_a enables a transition of B with target l_b then every location of A^* is reachable from l_a without visiting l_a again, while B is in l_b.*

Then $Reachloc(A^* \| B) = Reachloc(A_{l_a} \| B)$.

In contrast to all the other transformations, flattening in a parallel context does not easily generalize to multiple parallel instances (cf. Appendix C of [11]), and the three itemized conditions of Theorem 7 above require an exploration of the reachable state space. Such an exploration however does not entail a complete resolution of the $\|$ operator in $A^* \| B$, as each of the above conditions reduces to a *local reachability check of A^* resp. B*, with control residing at a fixed location

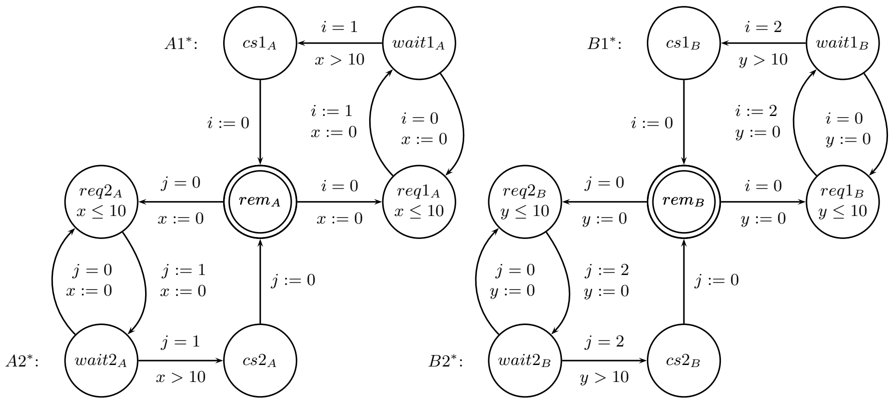


Fig. 1. Double Fischer protocol $DF = (A1^* \cup [A2^*]) \parallel (B1^* \cup [B2^*])$ for processes A and B accessing two critical sections $cs1$ and $cs2$. *Left*: ETA $A1^* \cup [A2^*]$; *right*: ETA $B1^* \cup [B2^*]$. In this and all subsequent figures, we omit the ε-labels of all edges.

of B resp. A^*. For the case where B is itself composed of multiple parallel ETA, a limited resolution of $\parallel$ within B may be necessary in order to verify the second reachability condition of Theorem 7. The above reachability checks may nonetheless be *localized within a layer* if the flattening is performed subsequent to separation and layering, as will be shown in the next section.

5 Example: Real-Time Mutual Exclusion

Consider two processes A and B competing for two critical sections $cs1$ and $cs2$. We safeguard these sections by a *double Fischer protocol DF* obtained by taking for each process the union of two copies of Fischer's real-time protocol.

We represent DF by the following composition of ETA:

$$DF = (A1^* \cup [A2^*]) \parallel (B1^* \cup [B2^*]),$$

where $A1^*, A2^*$ are the two copies of Fischer's protocol used by process A, and $B1^*, B2^*$ the two copies used by process B, cf. Fig. 1. The stars * indicate the presence of cycles. In locations $cs1_A$ and $cs2_A$ process A accesses the critical sections $cs1$ and $cs2$, respectively, and in $cs1_B$ and $cs2_B$ process B does so. Initially, A and B need not access the critical sections and are busy with *remaining* activities in the initial locations rem_A and rem_B, whose conditions permit the wrapping of all ETA, and in particular $A2^*$ and $B2^*$. In $req1_A, req2_A$ and $req1_B, req2_B$ the processes A and B *request* access to $cs1, cs2$. The locations $wait1_A, wait2_A$ and $wait1_B, wait2_B$ represent *waiting* of A and B for $cs1, cs2$. The parallel ETA in DF use disjoint (but synchronous) clocks x and y, while sharing the data variables i and j that range over $0, 1, 2$ and initialized with 0.

These values indicate whether (0) none of the processes, (1) process A, or (2) process B wants to access $cs1$ or $cs2$, respectively. In these ETA, all edges are labelled by ε-actions. Synchronization between the ETA takes place via guards checking the values of the shared variables i and j.

We wish to prove that DF satisfies the *double mutual exclusion* property

$$DMX = \Box\neg(cs1_A \wedge cs1_B) \wedge \Box\neg(cs2_A \wedge cs2_B).$$

We simplify the verification task now by a series of structural transformations so that it finally becomes almost trivial.

1. Separation. We apply the separation transformation to DF and obtain two single versions of Fischer's protocol separated by an extra transition, cf. Fig. 2. This is possible due to $A1^* \not\leftharpoondown B2^*$ and $B1^* \not\leftharpoondown A2^*$. The result is

$$SDF = (A1^* \rhd [A2^*]) \,\|\, (B1^* \rhd [B2^*]).$$

By the Separation Theorem 5, we have $DF \equiv_r SDF$.

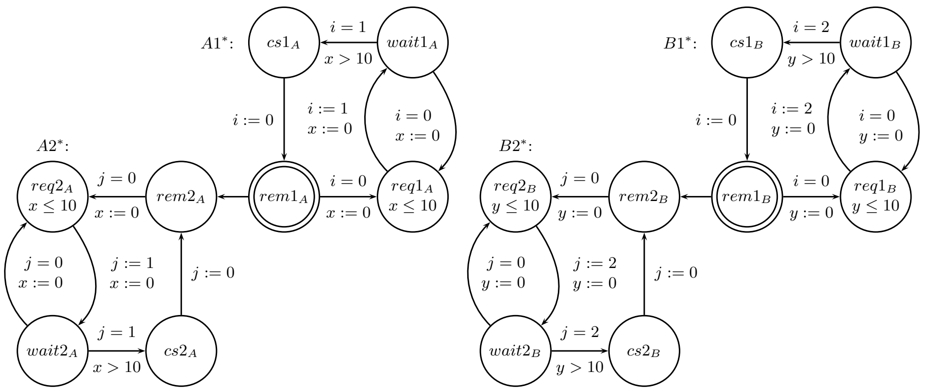


Fig. 2. Separated version of double Fischer: $SDF = (A1^* \rhd [A2^*]) \,\|\, (B1^* \rhd [B2^*])$. The layered version LDF is obtained from SDF by cutting the two component ETA of SDF at $rem2_A$ and $rem2_B$, yielding $LDF = (A1^* \,\|\, B1^*) \rhd ([A2^*] \,\|\, [B2^*])$.

2. Layering. To apply layering to SDF, we calculate:

$$\begin{aligned} SDF = \quad & (A1^* \rhd [A2^*]) \,\|\, (B1^* \rhd [B2^*]) \\ \equiv_L \quad & \{\text{ Corollary 1, using } A1^* \not\leftharpoondown B2^* \text{ and } B1^* \not\leftharpoondown A2^*\} \\ & (A1^* \,\|\, B1^*) \rhd ([A2^*] \,\|\, [B2^*]) \quad = \quad LDF \end{aligned}$$

LDF stands for layered double Fischer. The component ETA of LDF are obtained from the ETA shown in Fig. 2 by cutting these at $rem2_A$ and $rem2_B$.

To prove that DF satisfies DMX, it suffices to do this for LDF. Since step composition is the top operator in LDF, it suffices to show that both step components, $A1^* \| B1^*$ and $[A2^*] \,\|\, [B2^*]$, individually satisfy DMX.

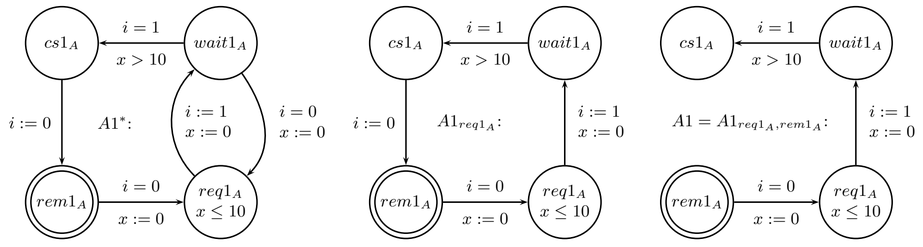


Fig. 3. Flattening $A1^*$ at location $req1_A$ yields $A1_{req1_A}$ and flattening this ETA at location $rem1_A$ yields the cycle-free ETA $A1 = A1_{req1_A,rem1_A}$

3. Flattening. We remove all cycles in $A1^*, B1^*, A2^*, B2^*$ and show this in detail for $A1^*$ in Fig. 3. Flattening $A1^*$ at $req1_A$ is possible, since whenever $req1_A$ is entered, $i = 0$ and $x = 0$ holds. Flattening the resulting $A1_{req1_A}$ at $rem1_A$ does not seem possible at first sight because only the data variable i is reset to 0. However, we may safely add $x := 0$ because this clock reset occurs when $rem1_A$ is left. Note also that the additional three conditions of Theorem 7 hold for $A1^*$ at the locations $req1_A$ and $rem1_A$. Hence, we arrive at $A1 = A1_{req1_A,rem1_A}$ without any cycles. We may similarly flatten $B1^*$ at $req1_B$ and $rem1_B$, yielding a corresponding cycle-free ETA $B1$. It thus remains to show that two cycle-free versions of Fischer's protocol, $A1 \parallel B1$ and $A2 \parallel B2$, individually satisfy *DMX*, where we have, for $i \in \{1, 2\}$, $\mathit{Reachloc}(Ai \| Bi) = \mathit{Reachloc}(Ai^* \| Bi^*)$ by Theorem 7, which is sufficient for preserving *DMX*.

4. Timed Layering. We prove that $A1 \| B1$ satisfies *DMX* (and symmetrically for $A2 \| B2$). Consider the ETA A_{01}, A_{11}, A_{21}, B_{01}, B_{11}, B_{21} shown in Fig. 4. As before, x and y are clocks, and i is a shared data variable ranging over 0, 1, 2, initialized with 0. The ETA A_{01}, A_{11}, A_{21} represent three phases of $A1$ and B_{01}, B_{11}, B_{21} those of $B1$, such that $A1 \| B1 = (A_{01}; A_{11}; A_{21}) \parallel (B_{01}; B_{11}; B_{21})$.

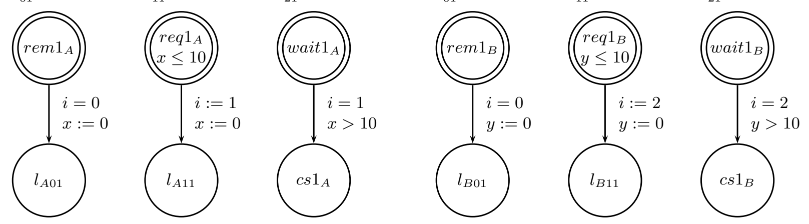


Fig. 4. Six ETA for building Fischer's protocol for single mutual exclusion of two processes: $A1 = A_{01}; A_{11}; A_{21}$ and $B1 = B_{01}; B_{11}; B_{21}$.

We explore the interleavings of $A_{01}; A_{11}; A_{21}$ with $B_{01}; B_{11}; B_{21}$ by (partially) *expanding* (as in CCS, cf. [14]) the parallel composition in $A1 \| B1$. After A_{11}

neither B_{01} nor B_{21} can occur due to the i-guard, and vice versa, after B_{11} neither A_{01} nor A_{21} can occur. After $A_{01} \| B_{01}$ we observe that timewise (by the synchronous evolution of the clocks x and y) B_{21} cannot proceed from $wait1_B$ to $cs1_B$ (due to the clock guard $y > 10$) before A_{11} has left $req1_A$ (with clock invariant $x \leq 10$) to reach its final location l_{A11}, and vice versa, A_{21} cannot proceed to $cs1_A$ before B_{11} reaches its final location l_{B11}. So the time precedences $A_{11} \prec B_{21}$ and $B_{11} \prec A_{21}$ hold. Thus expansion and Corollary 2 yield

$$\begin{aligned} A1 \| B1 &= (A_{01}; A_{11}; A_{21}) + (B_{01}; B_{11}; B_{21}) + (A_{01} \| B_{01}); ((A_{11}; A_{21}) \| (B_{11}; B_{21})) \\ &\equiv (A_{01}; A_{11}; A_{21}) + (B_{01}; B_{11}; B_{21}) + (A_{01} \| B_{01}); (A_{11} \| B_{11}); (A_{21} \| B_{21}) = SF, \end{aligned}$$

where + denotes a non-deterministic choice operator on ETA (cf. Definition 18 in Appendix A of [11]), and SF stands for sequential Fischer. Clearly, SF satisfies *DMX* because after $A_{11} \| B_{11}$ the data variable i stores either 1 or 2, and thus either A_{21} or B_{21} (but not both) can proceed to their critical section. Since each of the equivalences induced by our transformations (namely, $\equiv$, $\equiv_r$, $\equiv_L$, and equality w.r.t *Reachloc*) is clearly sufficient for *DMX*, we then conclude that *DMX* holds for DF as required.

6 Related Work

We now discuss related transformational approaches in the literature.

A constraint-based decompositional proof methodology was illustrated in [17] on the standard Fischer's protocol, formalized as a *timed modal specification.* More recently, an analysis of TA networks with "disjoint phases of activity" has been carried out in [18], where it has been shown that the parallel composition of two TA (without shared data variables) is bismilar to their sequential composition, if the TA exhibit certain *periodic but non-overlapping* behaviours.

In [19] it was shown that any TA (possibly containing nested cycles, but again without shared data variables) may be transformed into one that is flat (in the sense that each location is part of at most one cycle), while preserving the reachability relation between states. Their (non-local) transformation, while applicable to all TA, is however not preserved in the context of parallel composition, and suffers from an exponential blow-up in the number of locations in the resulting flattened TA, cf. Lemma 3 of [19]. Our (local) separation and flattening transformations, on the other hand, are applicable (in the context of parallel composition) to the data-enriched setting of ETA networks, and maintain the same number of locations, while reducing the nesting depth and deleting those transitions that (re-)enter memoryless locations, cf. Theorems 5 and 7.

A layered transformation for distributed algorithms with (predominantly synchronous) *message passing* was presented in [20]. *Round-based communication closedness* was considered in [21] for fault-tolerant distributed algorithms with *asynchronous message passing*, with messages being considered only in the rounds during which they were sent. Consensus algorithms in such a setting were then brought under the scope of automatic verification, by means of "reduction theorems", cf. [21]. Layered transformations for randomized distributed algorithms (modelled as compositions of probabilistic automata with shared data variables) have been recently investigated by the authors in [22].

Acknowledgements. A. Kupriyanov and the reviewers gave insightful feedback.

References

1. Elrad, T., Francez, N.: Decomposition of distributed programs into communication-closed layers. Sci. Comput. Program. 2, 155–173 (1982)
2. Janssen, W.: Layered Design of Parallel Systems. PhD thesis, U. Twente (1994)
3. Janssen, W., Poel, M., Xu, Q., Zwiers, J.: Layering of real-time distributed processes. In: Langmaack, H., de Roever, W.-P., Vytopil, J. (eds.) FTRTFT 1994 and ProCoS 1994. LNCS, vol. 863, pp. 393–417. Springer, Heidelberg (1994)
4. Olderog, E.-R., Swaminathan, M.: Layered composition for timed automata. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 228–242. Springer, Heidelberg (2010)
5. Alur, R., Dill, D.: A theory of timed automata. TCS, 183–235 (1994)
6. Bengtsson, J., Jonsson, B., Lilius, J., Yi, W.: Partial order reductions for timed systems. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 485–500. Springer, Heidelberg (1998)
7. Minea, M.: Partial order reduction for model checking of timed automata. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 431–436. Springer, Heidelberg (1999)
8. Lugiez, D., Niebert, P., Zennou, S.: A partial order semantics approach to the clock explosion problem of timed automata. Theor. Comput. Sci. 345, 27–59 (2005)
9. Håkansson, J., Pettersson, P.: Partial order reduction for verification of real-time components. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 211–226. Springer, Heidelberg (2007)
10. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
11. Olderog, E.R., Swaminathan, M.: Structural transformations for data-enriched real-time systems. Technical Report 90, Reports of SFB/TR 14 AVACS (2013), `http://www.avacs.org`
12. Bouyer, P., Petit, A.: Decomposition and composition of timed automata. In: Wiedermann, J., Van Emde Boas, P., Nielsen, M. (eds.) ICALP 1999. LNCS, vol. 1644, pp. 210–219. Springer, Heidelberg (1999)
13. Dong, J.S., Hao, P., Qin, S., Sun, J., Yi, W.: Timed automata patterns. IEEE Trans. Software Eng. 34, 844–859 (2008)
14. Milner, R.: Communication and Concurrency. Prentice-Hall (1989)
15. Alur, R., Brayton, R.K., Henzinger, T.A., Qadeer, S., Rajamani, S.K.: Partial-order reduction in symbolic state-space exploration. FMSD 18, 97–116 (2001)
16. Cohen, E.: Separation and reduction. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 45–59. Springer, Heidelberg (2000)

17. Larsen, K.G., Steffen, B., Weise, C.: Fischer's protocol revisited: A simple proof using modal constraints. In: Alur, R., Sontag, E.D., Henzinger, T.A. (eds.) HS 1995. LNCS, vol. 1066, pp. 604–615. Springer, Heidelberg (1996)
18. Muñiz, M., Westphal, B., Podelski, A.: Timed automata with disjoint activity. In: Jurdziński, M., Ničković, D. (eds.) FORMATS 2012. LNCS, vol. 7595, pp. 188–203. Springer, Heidelberg (2012)
19. Comon, H., Jurski, Y.: Timed automata and the theory of real numbers. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 242–257. Springer, Heidelberg (1999)
20. Stomp, F.A., de Roever, W.P.: A principle for sequential reasoning about distributed algorithms. Formal Asp. Comput. 6, 716–737 (1994)
21. Chaouch-Saad, M., Charron-Bost, B., Merz, S.: A reduction theorem for the verification of round-based distributed algorithms. In: Bournez, O., Potapov, I. (eds.) RP 2009. LNCS, vol. 5797, pp. 93–106. Springer, Heidelberg (2009)
22. Swaminathan, M., Katoen, J.P., Olderog, E.R.: Layered reasoning for randomized distributed algorithms. Formal Asp. Comput. 24, 477–496 (2012)

Deadlock Analysis of Concurrent Objects: Theory and Practice[⋆]

Elena Giachino[1], Carlo A. Grazia[1], Cosimo Laneve[1], Michael Lienhardt[2], and Peter Y. H. Wong[3]

[1] University of Bologna – INRIA Focus Team, Italy
[2] PPS, Paris Diderot, France
[3] SDL Fredhopper, Amsterdam, The Netherlands

Abstract. We present a framework for statically detecting deadlocks in a concurrent object language with asynchronous invocations and operations for getting values and releasing the control. Our approach is based on the integration of two static analysis techniques: (i) an inference algorithm to extract abstract descriptions of methods in the form of behavioral types, called contracts, and (ii) an evaluator that computes a fixpoint semantics returning a finite state model of contracts. A potential deadlock is detected when a circular dependency is found in some state of the model. We discuss the theory and the prototype implementation of our framework. Our tool is validated on an industrial case study based on the Fredhopper Access Server (FAS) developed by SDL Fredhoppper. In particular we verify one of the core concurrent components of FAS to be deadlock-free.

1 Introduction

Modern systems are designed to support a high degree of parallelism by ensuring that as many system components as possible are operating concurrently. Deadlock represents an insidious and recurring threat when such systems also exhibit a high degree of resource and data sharing. In these systems, deadlocks arise as a consequence of exclusive resource access and circular wait for accessing resources. A standard example is when two processes are exclusively holding a different resource and are requesting access to the resource held by the other. That is, the correct termination of each of the two process activities *depends* on the termination of the other. The presence of a *circular dependency* makes termination impossible.

Deadlocks may be particularly hard to detect in systems where the basic communication operation is asynchronous and where a synchronization would explicitly occur when the value is strictly needed. Further difficulties arise in the presence of unbounded (mutual) recursion. A paradigm case is an adaptive system that creates an unbounded number of processes such as server applications. In such systems, process interaction becomes complex and difficult to predict.

`ABS` [2] is an abstract, executable, object-oriented modeling language with a formal semantics, targeting distributed systems. The concurrency model of `ABS` is two-tiered. At the lower level it is similar to that of JCoBox [19], which in turn generalizes

⋆ Partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (`http://www.hats-project.eu`).

the concurrency model of Creol [14] from single concurrent objects to concurrent object groups (COGs). COGs encapsulate synchronous, multi-threaded, shared state computation on a single processor. On top of this level, there is an actor-based model with asynchronous calls, message passing, active waiting, and future types.

An essential difference to thread-based concurrency is that task scheduling is *cooperative*, i.e., control switching between tasks of the same object group happens only at specific scheduling points during the execution, which are explicit in the source code and can be syntactically identified. This allows one to write concurrent programs in a much less error-prone way than in a thread-based model and makes `ABS` models suitable for static analysis.

We developed a theoretical framework for statically detecting deadlocks in `core ABS` [13] (a subset of `ABS`) programs, exploiting and combining results and techniques coming from different well-known theories:

Type theory. We designed an inference system to automatically extract abstract behavioral descriptions pertinent to deadlock analysis from `core ABS` code. These descriptions are called *contracts*. This is necessary as analyzing the whole program would be hard and time-consuming, while most part of the code would be irrelevant for deadlock and synchronization behavior, such as the local data and computations. The inference system is constraint-based and uses a standard semiunification technique for solving the set of generated constraints.

Abstract behavioral language. Contracts are defined by a basic behavioral language, that is similar to those ranging from languages for session types to calculi of processes as Milner's CCS or pi-calculus. There are a wide number of theories and tools for verifying their properties. However, unlike most techniques on deadlock analysis, our behavioral language handles dynamic name creation and does not require a predefined partial order.

Fixpoint Theory. The semantics of contracts is denotational. However, in presence of recursion in the code, a fixpoint may not exist because the underlying model would have infinitely many states (due to the creation of new objects in recursive methods). To circumvent this issue, we use a fixpoint technique on models with a limited capacity of name creation. This entails fixpoint existence and finiteness of models. While we lose precision, our technique is sound (in some cases, our technique may signal false positives).

We prototyped an implementation of our framework, called the SDA tool and validated it via an industrial case study. The case study is based on the Fredhopper Access Server (FAS) developed by SDL Fredhopper[1]. In particular we were able to verify one of the core concurrent components of FAS to be deadlock-free.

The structure of the paper is as follows. In Section 2, we introduce the `core ABS` language, emphasizing its concurrency model, which is most relevant to this paper. In Section 3, we present the behavioral language for specifying contracts and the inference system for extracting contracts from `core ABS` programs. In Section 4, we overview the algorithm for computing the contracts into their associated abstract models. In Section 5 we present the implementation of the tool, and its validation against the case study.

[1] `http://sdl.com/products/fredhopper/`

Related works are discussed in Section 6 and final remarks are collected in Section 7. Due to space limitations, some technical parts are informally discussed and proofs are omitted; they can be found in the full paper.

2 The Language `core ABS`

We begin with a brief presentation of the syntax of `core ABS` (See Figure 1). Full details of the language, its semantics and its type system, can be found in [13].

In the syntax, an overlined element corresponds to any finite sequence of such element; as usual, an element between square brackets is optional. When we write $\overline{T}$ (or $\overline{V}$ or $\overline{T\ x}$ or $\overline{e}$ or $\overline{v}$) we mean a (possibly empty) sequence $T_1, \cdots, T_n$ (or, respectively, $V_1, \cdots, V_n$ or $T_1\ x_1, \cdots, T_n\ x_n$ or $e_1, \cdots, e_n$ or $v_1, \cdots, v_n$).

A program P is a list of declarations followed by a main function $\{\,\overline{T\ x\ ;}\ \ s\,\}$. Declarations include *data-types* D and *functions* F, which constitute the functional part of the language, and *interfaces* I and *classes* C, which constitute the object-oriented part of the language. A type T is the name of either a type variable `V` used for polymorphism, a datatype with parameters $\mathtt{D}\langle\overline{T}\rangle$ (to type structured data), or an interface `I` (to type objects). A data type D has a name `D` and a sequence of parameters $\overline{V}$, and is constructed as a nonempty sequence of type constructors `Co` with possible parameters $\overline{T}$. Note that data types include primitive types such as `Int`, `Bool`, `String`, as well as complex types such as list of integers $\mathtt{List}\langle\mathtt{Int}\rangle$. The complex type $\mathtt{Fut}\langle T\rangle$ is called *future type* and its values are *futures*. The future type is relevant in `core ABS` because it is used to type method invocations (that return values of type T). A function F has a return type T, a name `f`, an optional sequence of type parameters $\overline{T}$ (for polymorphism), a sequence of parameters $\overline{T\ x}$, and returns the value of the expression e. An interface I has a name `I` and a body declaring a sequence of method headers S. A class C has a name `C`, may implement several interfaces, and declares its fields Fl and its methods M.

A statement s may be either one of the standard operations of a core imperative language or one of the operations for scheduling. Scheduling operations include `await`

$$\begin{array}{rcll}
P & ::= & \overline{D}\ \overline{F}\ \overline{I}\ \overline{C}\ \{\,\overline{T\ x\ ;}\ \ s\,\} & \text{program}\\
T & ::= & \mathtt{V}\ \mid\ \mathtt{D}\langle\overline{T}\rangle\ \mid\ \mathtt{I} & \text{type}\\
D & ::= & \mathtt{data}\ \mathtt{D}\langle\overline{V}\rangle = \overline{\mathtt{Co}\,[\,(\overline{T})\,]} & \text{data type}\\
F & ::= & \mathtt{def}\ T\ \mathtt{f}\ [\langle\overline{T}\rangle](\overline{T\ x}) = e & \text{function}\\
I & ::= & \mathtt{interface}\ \mathtt{I}\,\{\,\overline{S\ ;}\,\} & \text{interface}\\
S & ::= & T\ \mathtt{m}(\overline{T\ x}) & \text{method signature}\\
C & ::= & \mathtt{class}\ \mathtt{C}(\overline{T\ x})\ [\mathtt{implements}\ \overline{\mathtt{I}}]\,\{\,\overline{T\ x\ ;}\ \ \overline{M}\,\} & \text{class}\\
M & ::= & S\{\,\overline{T\ x\ ;}\ \ s\,\} & \text{method definition}\\
s & ::= & \mathtt{skip}\ \mid\ s\ ;\ s\ \mid\ x = z\ \mid\ \mathtt{await}\ g & \text{statement}\\
 & \mid & \mathtt{if}\ e\,\{\,s\,\}\ \mathtt{else}\,\{\,s\,\}\ \mid\ \mathtt{while}\ e\,\{\,s\,\}\ \mid\ \mathtt{return}\ e & \\
z & ::= & e\ \mid\ \mathtt{new}\ [\mathtt{cog}]\ \mathtt{C}\,(\overline{e})\ \mid\ e.\mathtt{m}(\overline{e})\ \mid\ e!\mathtt{m}(\overline{e})\ \mid\ e.\mathtt{get} & \text{expression with side effects}\\
e & ::= & v\ \mid\ x\ \mid\ \mathtt{this}\ \mid\ \mathtt{fun}(\overline{e})\ \mid\ \mathtt{case}\ e\,\{\overline{p \Rightarrow e}\} & \text{expression}\\
v & ::= & \mathtt{null}\ \mid\ \mathtt{Co}[(\overline{v})] & \text{value}\\
p & ::= & \_\ \mid\ x\ \mid\ \mathtt{null}\ \mid\ \mathtt{Co}[(\overline{p})] & \text{pattern}\\
g & ::= & e\ \mid\ x?\ \mid\ g \wedge g & \text{guard}
\end{array}$$

Fig. 1. The language `core ABS`

g, which suspends the method's execution until the argument, called a *guard*, becomes true. Guards may be a Boolean expression e that must be true in order to continue the method's execution, and a future lookup x? that requires the value of x to be resolved before resuming the method's execution, or a conjunction of guards $g \wedge g$.

An expression z may have side effects (may change the state of the system) and is either an object creation `new C` ($\overline{e}$) in the same group of the creator or an object creation `new cog C` ($\overline{e}$) in a new group, a method call e.`m`($\overline{e}$) or e!`m`($\overline{e}$), or a `get` on a expression returning a future value.On the other hand, a *pure* expression e is free of side effects and is either a value v, a variable x, a function application `fun`($\overline{e}$), or a pattern matching `case` e $\{\overline{p \Rightarrow e}\}$. Values include the `null` object, and structured data `Co`[($\overline{v}$)], while patterns p extend these values with variables x and anonymous variables _.

2.1 The Concurrency Model of `core ABS`

We describe informally the concurrency model of `core ABS` and provide an illustration in the form of a small example. In `core ABS`, objects belong to a group; a task executing an object method belongs to the object's group. At each point in time there is at most one task per group that is active. The active task must explicitly release control in order for another task of the same group to progress. Tasks are created by method invocations: the caller activity continues after the invocation while the called code runs as a new task. Caller and callee synchronize when the returned value of the method is strictly necessary. In order to decouple method call and returned value, `core ABS` uses futures, i.e., pointers to returned values that may not yet be available. Accesses to the future values may require waiting for the values to be returned.

The code in Figure 2 gives three different implementations of the factorial function in an hypothetical class `Math`. The function `fact_g` is the standard definition of factorial: the recursive invocation `this!fact_g(n-1)` is followed by a `get` operation that retrieves the value returned by the invocation. Yet, `get` does not allow the task to release the group lock; therefore the task evaluating `this!fact_g(n-1)` is fated to be delayed forever because its object (and, therefore, the corresponding group) is the same as that of the

```
class Math {
  Int fact_g(Int n){
    if (n==0) { return 1; }
    else { Fut<Int> x = this!fact_g(n-1); Int m = x.get; return n*m; } }
  Int fact_ag(Int n){
    if (n==0) { return 1; }
    else { Fut<Int> x = this!fact_ag(n-1); await x?; Int m = x.get;
           return n*m; } }
  Int fact_nc(Int n){
    if (n==0) { return 1 ; }
    else { Math z = new cog Math(); Fut<Int> x = z!fact_nc(n-1);
           Int m = x.get; return n*m; } } }
```

Fig. 2. The class `Math`

caller. The function `fact_ag` solves this problem by permitting the caller to release the lock with an explicit `await` operation, before getting the actual value with `x.get`. An alternative solution is defined by the function `fact_nc`, whose code is similar to that of `fact_g`, except for that `fact_nc` invokes `z!fact_nc(n-1)` recursively, where `z` is an object in a new group. This means the task of `z!fact_nc(n-1)` may start without waiting for the release of any lock by the caller.

2.2 Restrictions of `core ABS` of the Current Release of SDA

In order to verify the feasibility of our techniques, in the first release of our prototype we considered a subset of `core ABS` features. Note that these restrictions have been considered in order to ease the initial development of the SDA tool. These restrictions do not jeopardize the tool's extension to the full language. Below we discuss the restrictions and, for each of them, we detail the techniques that will be used to remove them in the next release of SDA. We also notice that, notwithstanding the following restrictions, we were able to verify large commercial codes, such as a core component of FAS (Section 5.2).

Split Synchronizations. `core ABS` allows synchronization primitives (`await` and `get`) to be performed long after the method invocation. Recording the associated invocation-synchronization primitives is problematic because it requires the analysis of aliases. To avoid such complexity, we constrain codes to perform the synchronization, when needed, right after the method invocation. Clearly, the extension of the SDA tool with a standard alias analysis will permit the removal of this constraint.

Synchronization on Booleans. In addition to synchronization on method invocations, `core ABS` permits synchronizations on Booleans, with the statement `await` g. When g is `False`, the execution of the method is suspended, and when it becomes `True`, the `await` terminates and the execution of the method may proceed. It is possible that the expression g refers to a field of an object that can be modified by another method. In this case, the `await` becomes synchronized with any method that may set the field to `true`. This subtle synchronization pattern is difficult to verify statically. We therefore require `await` statements to be annotated with the dependencies they create. For example, consider the code:

```
class ClientJob(...) {
 Schedules schedules = EmptySet; ConnectionThread thread; ...
 Unit executeJob() {
   thread = ...; thread!command(ListSchedule);
   [thread] await schedules != EmptySet; ... }}
```

The statement `await` compels the task to wait for `schedules` to be set to something different from the empty set. Since `schedules` is a field of the object, any concurrent thread (on that object) may update it. It is not evident how to extract this implicit dependency relation from the guard of `await`. Therefore we constrain the programmer to provide an annotation making explicit the dependency. In the above case, the object that will modify the boolean guard is stored in the variable `thread`. Thus we need the annotation `[thread]`.

Data Types and While Loops. In `core ABS`, data types are used to define primitive types (e.g. Booleans) and dynamic structures, such as lists or maps. In particular, dynamic structures can store an unbounded number of objects and, using a `while` loop, it is possible to invoke methods on these objects according to some ad-hoc protocol. This is problematic as our technique concerns static analysis. As a result we require that: *(i)* data types are simply used to store objects of the same class; *(ii)* at each iteration, these objects are manipulated independently (no synchronization with objects in the context is performed), and in an identical manner. A `core ABS` program with these properties may be analyzed for deadlocks using *representatives*. Namely, a data type value is abstracted by one of its objects and a `while` loop is abstracted by its body. Note that both conditions hold in many usages of dynamic data types and iteration, particularly in the case study. The next release of SDA will permit ad-hoc annotations for `while` loops (invariants) that affect contracts generated by the inference system.

Assignments and Local Variables. Assignments in `core ABS` (as usual in object-oriented languages) may update the fields of objects that are accessed concurrently by other threads, thus could lead to indeterminate behavior. In order to simplify the analysis, we constrain field assignments to keep field's record structure unchanged. For instance, if a field contains an object of group a, then that field may be only updated with objects belonging to a (and this correspondence must hold recursively with respect to the fields of objects referenced by a). When the field is of a primitive type (`Int`, `Bool`, etc.) this constraint is equivalent to the standard type-correctness. This restriction does not cover local variables of methods, as they can only be accessed by the method in which they are declared. In fact it is easy to track local changes in the inference algorithm. It is possible to be more liberal as regards fields assignments. In [11] an initial study for covering full-fledged field assignments was undertaken using so-called union types (that is, by extending the syntax of future records with a + operator, as for contracts, see below) and collecting all records in the inference rule of the field assignment (and the conditional).

Interfaces. In `core ABS` objects are typed with interfaces, which may have several implementations. As a consequence, when a method is invoked, it is in general not possible to statically determine which method will be executed at runtime (dynamic dispatch). This is problematic for our technique because it breaks the association of a unique abstract behavior with a method invocation. In the current release of SDA we avoid this issue by constraining codes to have interfaces implemented by at most one class. This restriction will be relaxed by admitting that methods have multiple contracts, one for every possible class implementation of the arguments, and return values of methods are unions of records. In turn, method invocations yield unions of contracts, according to the possible instantiations of their arguments.

Recursive Object Structures. In `core ABS`, like in any other object-oriented language, it is possible to define circular object structures, such as an object storing a pointer to itself in one of its fields. Currently, the SDA tool cannot deal with recursive structures, because the semi-unification process associates each object with a finite tree structure. In this way, it is not possible to capture circular definitions, such as the recursive ones. This restriction will be removed in the next release of SDA by admitting the association of *regular terms* [5] with objects in the semi-unification process.

3 Contracts and the Contract Inference System

In order to analyze `core ABS` codes, we use abstract descriptions called *contracts*. The syntax of these descriptions uses *record names* X, Y, Z, ..., and *group names* a, b, ... The rules are

Future records $\mathbb{r}$ encode the values of expressions in contracts. A record may be one of the following: an empty record $\_$, which corresponds to primitive types; a record name X, which represents a place-holder for a value and can be instantiated by substitutions; $a[\overline{\mathtt{f}} : \overline{\mathbb{r}}]$ that defines an object with its group name a, and $a \rightsquigarrow \mathbb{r}$ which specifies that accessing $\mathbb{r}$ requires control of the group a (and that the control is to be released once the method has been evaluated). Note that the future record $a \rightsquigarrow \mathbb{r}$ is associated with method invocations: a is the group of the object on which the method is invoked.

Contracts $\mathbb{c}$ collect the method invocations and the group dependencies inside statements. Apart from $\mathbb{0}$, (a, a'), and $(a, a')^{\mathtt{w}}$ that respectively represent the empty behavior, the dependency pairs due to a `get` and an `await` operation, the other basic contracts deal with method invocations. The contract $\mathtt{C.m}\, \mathbb{r}(\overline{\mathbb{r}}) \rightarrow \mathbb{r}'$ models synchronous method invocations, while $\mathtt{C!m}\, \mathbb{r}(\overline{\mathbb{r}}) \rightarrow \mathbb{r}'$ models asynchronous invocations. This latter contract may be followed by a `get` – the suffix "$\centerdot (a, a')$" –, or followed by an `await` – the suffix "$\centerdot (a, a')^{\mathtt{w}}$". Composite contracts define the sequential composition $\mathbb{c} \fatsemi \mathbb{c}'$ and conditionals $\mathbb{c} + \mathbb{c}'$.

Finally, our tool uses constraints $\mathcal{U}$ that are defined by the following syntax

where `true` is the constraint that is always true; $\mathbb{r} = \mathbb{r}'$ is a classic unification constraint between terms; $\mathbb{r}(\overline{\mathbb{r}}) \rightarrow \mathbb{s} \preceq \mathbb{r}'(\overline{\mathbb{r}'}) \rightarrow \mathbb{s}'$ is a *semiunification* constraint; the constraint $\mathcal{U} \wedge \mathcal{U}'$ is the conjunction of $\mathcal{U}$ and $\mathcal{U}'$.

Contracts are extracted from `core ABS` programs by means of an inference algorithm. Figures 3 and 4 illustrate a (relevant) subset of the rules; the other ones are omitted to lighten our presentation. The following auxiliary operators are used: *fields*(`C`) and *param*(`C`) return the sequence of fields and parameters of a class `C`; *types*(e) returns the type of an expression e, which is an interface; if e is an object, *class*(I) returns the unique (see the restriction *Interfaces* in Section 2.2) class implementing I; and *mname*($\overline{M}$) returns the sequence of method names in the sequence $\overline{M}$ of method declarations.

Inference statements for pure expressions e have the form $\Gamma \vdash_a e : \mathbb{r}$, where Γ is a typing context mapping variables to their records, and methods to their signatures; a is the group name of the object executing the expression; and $\mathbb{r}$ is the inferred record. Constraints and contracts are not generated at this stage.

Inference statements for expressions z have the form $\Gamma \vdash_a z : \mathbb{r}, \mathbb{c} \rhd \mathcal{U}$ where Γ, a, and $\mathbb{r}$ are as for expressions e. The term $\mathbb{c}$ is the contract for z created by the inference rules and $\mathcal{U}$ is the generated constraint. The rule NewCOG creates a new group name that is returned in the record of the expression, while New uses the name of the group of `this`. It is worth to recall that, in `core ABS`, the creation of an object, either with a `new` or with a `new cog`, amounts to executing the method `init` of the corresponding class, whenever defined (the `new` performs a synchronous invocation, the `new cog` performs an asynchronous one). In turn, the termination of `init` triggers the execution of the method `run`, if present. The method `run` is asynchronously invoked when `init` is absent. Since `init` may be regarded as a method in `core ABS`, the inference system in our tool explicitly introduces a synchronous invocation to `init` in case of `new` and an asynchronous

$$\textsc{Var}\quad \frac{x \in dom(\Gamma)}{\Gamma \vdash_a x : \Gamma(x)} \qquad \textsc{Field}\quad \frac{x \notin dom(\Gamma) \qquad \Gamma(\mathtt{this}) = a[\mathtt{f}' : \mathfrak{r}; \overline{\mathtt{f}} : \overline{\mathfrak{r}}]}{\Gamma \vdash_a \mathtt{f}' : \mathfrak{r}} \qquad \textsc{Get}\quad \frac{\Gamma \vdash_a e : \mathfrak{r} \qquad X, b \text{ fresh}}{\Gamma \vdash_a e.\mathtt{get} : X, (a,b) \rhd \mathfrak{r} = b \rightsquigarrow X}$$

$$\textsc{NewCog}\quad \frac{\begin{array}{c}\Gamma \vdash_a \overline{e} : \overline{\mathfrak{r}} \qquad a' \text{ fresh}\\ fields(\mathtt{C}) = \overline{\mathtt{f}} \qquad param(\mathtt{C}) = \overline{\mathtt{f}}' \qquad \overline{X} \text{ fresh}\end{array}}{\Gamma \vdash_a \mathtt{new\ cog\ C}(\overline{e}) : a'[\overline{\mathtt{f}} : \overline{X}; \overline{\mathtt{f}}' : \overline{\mathfrak{r}}], \mathbb{0} \rhd \mathtt{true}} \qquad \textsc{New}\quad \frac{\begin{array}{c}\Gamma \vdash_a \overline{e} : \overline{\mathfrak{r}} \qquad \overline{X} \text{ fresh}\\ fields(\mathtt{C}) = \overline{\mathtt{f}} \qquad param(\mathtt{C}) = \overline{\mathtt{f}}'\end{array}}{\Gamma \vdash_a \mathtt{new\ C}(\overline{e}) : a[\overline{\mathtt{f}} : \overline{X}; \overline{\mathtt{f}}' : \overline{\mathfrak{r}}], \mathbb{0} \rhd \mathtt{true}}$$

$$\textsc{AInvk}\quad \frac{\Gamma \vdash_a e : \mathfrak{r} \qquad \Gamma \vdash_a \overline{e} : \overline{\mathfrak{s}} \qquad class(types(e)) = \mathtt{C} \qquad b, Y, \overline{Y} \text{ fresh}}{\Gamma \vdash_a e!\mathtt{m}(\overline{e}) : b \rightsquigarrow Y, \mathtt{C!m}\ \mathfrak{r}(\overline{\mathfrak{s}}) \to Y \rhd b[\overline{\mathtt{f}} : \overline{Y}] = \mathfrak{r} \land \mathtt{C.m} \preceq \mathfrak{r}(\overline{\mathfrak{s}}) \to Y} \qquad \textsc{Return}\quad \frac{\Gamma \vdash_a e : \mathfrak{r} \qquad \Gamma(\mathtt{destiny}) = \mathfrak{s}}{\Gamma \vdash_a \mathtt{return}\, e : \mathbb{0} \rhd \mathfrak{r} = \mathfrak{s} \,|\, \Gamma}$$

$$\textsc{SInvk}\quad \frac{\begin{array}{c}\Gamma \vdash_a e : \mathfrak{r} \qquad \Gamma \vdash_a \overline{e} : \overline{\mathfrak{s}}\\ class(types(e)) = \mathtt{C} \qquad Y \text{ fresh}\end{array}}{\Gamma \vdash_a e.\mathtt{m}(\overline{e}) : a \rightsquigarrow Y, \mathtt{C.m}\ \mathfrak{r}(\overline{\mathfrak{s}}) \to Y \rhd \mathtt{C.m} \preceq \mathfrak{r}(\overline{\mathfrak{s}}) \to Y} \qquad \textsc{Await}\quad \frac{\Gamma \vdash_a x : \mathfrak{r} \qquad X, b \text{ fresh}}{\Gamma \vdash_a \mathtt{await}\, x? : (a,b)^{\mathtt{w}} \rhd \mathfrak{r} = b \rightsquigarrow X \,|\, \Gamma}$$

$$\textsc{Await-b}\quad \frac{\begin{array}{c}\Gamma \vdash_a x : \mathfrak{r} \qquad \overline{X}, b \text{ fresh}\\ class(types(x)) = \mathtt{C} \qquad fields(\mathtt{C}) = \overline{\mathtt{f}}\end{array}}{\Gamma \vdash_a [x]\,\mathtt{await}\, y : (a,b)^{\mathtt{w}}_{\natural} \rhd \mathfrak{r} = b[\overline{\mathtt{f}} : \overline{X}] \,|\, \Gamma} \qquad \textsc{AssignVar}\quad \frac{x \in dom(\Gamma) \qquad \Gamma \vdash_a z : \mathfrak{r}, \mathfrak{c} \rhd \mathcal{U}}{\Gamma \vdash_a x = z : \mathfrak{c} \rhd \mathcal{U} \,|\, \Gamma[x = \mathfrak{r}]} \qquad \textsc{AssignField}\quad \frac{\begin{array}{c}\Gamma \vdash_a z : \mathfrak{r}, \mathfrak{c} \rhd \mathcal{U} \qquad \mathtt{f}' \notin dom(\Gamma)\\ \Gamma(\mathtt{this}) = a[\mathtt{f}' : \mathfrak{r}'; \overline{\mathtt{f}} : \overline{\mathfrak{r}}]\end{array}}{\Gamma \vdash_a \mathtt{f}' = z : \mathfrak{c} \rhd \mathcal{U} \land \mathfrak{r} = \mathfrak{r}' \,|\, \Gamma}$$

$$\textsc{If}\quad \frac{\begin{array}{c}\Gamma \vdash_a e : \mathfrak{r} \qquad \Gamma \vdash_a s_1 : \mathfrak{c}_1 \rhd \mathcal{U}_1 \,|\, \Gamma_1\\ \Gamma \vdash_a s_2 : \mathfrak{c}_2 \rhd \mathcal{U}_2 \,|\, \Gamma_2 \qquad \Gamma_1|_{dom(\Gamma)} = \Gamma_2|_{dom(\Gamma)}\end{array}}{\Gamma \vdash_a \mathtt{if}\, e\, \{\, s_1 \,\}\, \mathtt{else}\, \{\, s_2 \,\} : \mathfrak{c}_1 + \mathfrak{c}_2 \rhd \mathcal{U}_1 \land \mathcal{U}_2 \,|\, \Gamma_1|_{dom(\Gamma)}} \qquad \textsc{Seq}\quad \frac{\begin{array}{c}\Gamma \vdash_a s_1 : \mathfrak{c}_1 \rhd \mathcal{U}_1 \,|\, \Gamma_1\\ \Gamma_1 \vdash_a s_2 : \mathfrak{c}_2 \rhd \mathcal{U}_2 \,|\, \Gamma_2\end{array}}{\Gamma \vdash_a s_1; s_2 : \mathfrak{c}_1 \between \mathfrak{c}_2 \rhd \mathcal{U}_1 \land \mathcal{U}_2 \,|\, \Gamma_2}$$

Fig. 3. Contract inference for expressions and statements

(Method)

$$\frac{\begin{array}{c} fields(\mathtt{C}) = \overline{\mathtt{f}} \quad param(\mathtt{C}) = \overline{\mathtt{f}}' \quad a, \overline{X}, \overline{Y}, Z\, fresh\\ \Gamma + \mathtt{this} : a[\overline{\mathtt{ff}'} : \overline{X}] + \overline{\mathtt{x}} : \overline{Y} + \mathtt{destiny} : Z \vdash_a s : \mathfrak{c} \rhd \mathcal{U} \,|\, \Gamma' \end{array}}{\Gamma \vdash \mathtt{T\ m}\ (\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{s\} \ :\ a[\overline{\mathtt{ff}'} : \overline{X}](\overline{Y})\{\mathfrak{c}\}\, Z \ \rhd\ \mathcal{U} \land a[\overline{\mathtt{ff}'} : \overline{X}](\overline{Y}) \to Z = \mathtt{C.m} \quad \text{IN } \mathtt{C}}$$

(Class)

$$\frac{\overline{X} \text{ fresh} \quad \Gamma + \overline{\mathtt{ff}'} : \overline{X} \vdash \overline{M} : \overline{\mathfrak{C}} \rhd \overline{\mathcal{U}} \quad \text{IN } \mathtt{C}}{\Gamma \vdash \mathtt{class\ C}(\overline{\mathtt{T}}\ \overline{\mathtt{f}})\ \{\overline{\mathtt{T}}'\ \overline{\mathtt{f}}';\ \ \overline{M}\} \ :\ \{mname(\overline{M}) \mapsto \overline{\mathfrak{C}}\} \ \rhd\ \overline{\mathcal{U}}}$$

Fig. 4. Contract rules of method and class declarations

one in case of `new cog`. However, for simplicity, we overlook this (simple) issue in the rules New and NewCog, acting as if `init` and `run` are always absent.

Rules for statements s have the form $\Gamma \vdash_a s : \mathfrak{c} \rhd \mathcal{U} \,|\, \Gamma'$ where Γ, a, s, $\mathfrak{c}$ and $\mathcal{U}$ are as before, and Γ' is the environment of the method after the execution of the statement. The environment may change because of local variable updates. Rule Await deals with the `await` synchronization when applied to a simple future lookup $x?$, returning a dependency $(a,b)^{\mathtt{w}}$. In order to correctly associate dependencies with each synchronization, we assume statements of the form `await` $(?x_1 \wedge ?x_2)$ to be decomposed into `await` $?x_1$ `;` `await` $?x_2$. Rule AssignVar manages assignments to local variables of methods and is the only rule that changes the environment. This rule must be compared with AssignField, which deals with assignment to fields. In this case, as we said before, since we do not admit field updates, the rule enforces that the future record of the right-hand-side expression to be the same as that of the field. Rule Return constrains the record of `destiny`, which is an identifier introduced by Method, shown in Figure 4, for storing the return record. Rule Seq defines the sequential composition of contracts. This rule uses an auxiliary binary operator $\between$ on contracts to manage accumulations of dependencies in sequence. The operator $\between$ is defined case-by-case. For example

$$\mathbb{c} \,;\, \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{s}}) \rightarrow \mathbb{r}' \between \mathbb{c}' = \begin{cases} \mathbb{c} \,;\, \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{s}}) \rightarrow \mathbb{r}' \centerdot (a,b) \,;\, \mathbb{c}'' & \text{if } \mathbb{c}' = (a,b) \,;\, \mathbb{c}'' \\ \mathbb{c} \,;\, \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{s}}) \rightarrow \mathbb{r}' \centerdot (a,b)^{\mathtt{w}} \,;\, \mathbb{c}'' & \text{if } \mathbb{c}' = (a,b)^{\mathtt{w}} \,;\, \mathbb{c}'' \\ \mathbb{c} \,;\, \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{s}}) \rightarrow \mathbb{r}' \,;\, \mathbb{c}' & \text{otherwise} \end{cases}$$

The rules for method and class declarations are defined in Figure 4. In Method, in order to derive the method contract of `T m` $(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{s\}$, we infer the type of s in an environment extended with `this`, `destiny` (that will be set by `return` statements), and the arguments $\overline{\mathtt{x}}$. The resulting contract $\mathbb{c}$ will be used in the method contract. The rule Class yields an *abstract class table* that associates a method contract with every method name. It is this abstract class table that is used by our analyzer in Section 4.

As an example, the methods of `Math` in Figure 2 have the following contracts, once the constraints are solved (we always simplify $\mathbb{c} \,;\, \mathbb{0}$ into $\mathbb{c}$):

- `fact_g` has contract $a[\,](\_)\ \{\mathbb{0} + \mathtt{Math!fact\_g}\ a[\,](\_) \rightarrow \_ \centerdot (a,a)\}\ \_$. The name a in the header refers to the group name associated with `this` in the code, and binds the occurrences of a in the body. The contract body has a recursive invocation to `fact_g`, which is performed on an object in the same group a and followed by a `get` operation. This operation introduces a dependency pair (a,a). We observe that, if we replace the statement `Fut<Int> x = this!fact_g(n-1)` in `fact_g` with `Math z = new Math() ; Fut<Int> x = z!fact_g(n-1)`, we obtain the same contract as above because the new object is in the same group as `this`.
- `fact_ag` has contract $a[\,](\_)\ \{\mathbb{0} + \mathtt{Math!fact\_ag}\ a[\,](\_) \rightarrow \_ \centerdot (a,a)^{\mathtt{w}}\}\ \_$. In this case, the presence of an `await` statement in the method body produces a dependency pair $(a,a)^{\mathtt{w}}$. The subsequent `get` operation does not introduce any dependency pair: (a,a) is absorbed by $(a,a)^{\mathtt{w}}$ by definition of $\between$. Intuitively, in this case, the success of `get` is guaranteed, provided the success of the `await` synchronization.
- `fact_nc` has contract $a[\,](\_)\ \{\mathbb{0} + \mathtt{Math!fact\_nc}\ b[\,](\_) \rightarrow \_ \centerdot (a,b)\}\ \_$. This method contract differs from the previous ones in that the receiver of the recursive invocation is a free name (i.e., it is not bound by a in the header). This because the recursive invocation is performed on an object of a new group (which is therefore different from a). As a consequence, the dependency pair added by the `get` relates the group a of `this` with the new group b.

Properties. The inference system for contracts possesses the classic soundness and completeness properties.

Theorem 1. *The inference system for contracts produces a class table (when the semi-unification algorithm terminates) that is sound and complete.*

This result is proved in a standard way by (1) defining a type system for contracts where method contracts are explicitly provided by programmers; then by (2) demonstrating that this type system is sound with respect to the operational semantics in [13] (subject reduction); and finally by (3) proving that the class table obtained by the inference system yields method contracts that are type correct with respect to (1) (completeness). As regards (1), the type system is very similar to the inference one, but it does not collect constraints. As regards (3), the rules also produce a set of semiunification constraints [12] $\mathtt{r}(\overline{\mathtt{r}}) \to \mathtt{s} \preceq \mathtt{r}'(\overline{\mathtt{r}'}) \to \mathtt{s}'$ by binding constraints of the form $\mathtt{r}(\overline{\mathtt{r}}) \to \mathtt{s} = \mathtt{C.m}$ (rule Method) with constraints of the form $\mathtt{C.m} \preceq \mathtt{r}'(\overline{\mathtt{r}'}) \to \mathtt{s}'$ (rules AInvk and SInvk). It is well-known that solving these constraints is undecidable in general [16]. Therefore, it is to be expected that the algorithm loops indefinitely in some cases, which are defined in very ad-hoc ways. In our various tests, we never reached this limitation of our approach.

4 The Analysis of Contracts

Contracts are inputs to our deadlock analysis technique, which returns finite state models, called *lam* (an acronym for deadLock Analysis Models [10]), where states are relations on group names. For example:

– $[\![(a,b)]\!], [\![(a,b),(b,c)]\!]$ is a two-states lam where one state contains the relation $\{(a,b)\}$ and the other state contains $\{(a,b),(b,c)\}$;
– $[\![(a,b)^{\mathtt{w}}]\!]$ is a one-state lam containing the relation $\{(a,b)^{\mathtt{w}}\}$.

The algorithm takes as input an abstract class table and a main contract, both produced by the inference system; then it applies the standard Knaster-Tarski technique. The critical issue of this technique is that it may create pairs on fresh names at each step, technically speaking, at *every approximant*, because of free names in method contracts that correspond to `new cog`s. As a consequence, the lam model *is not a complete partial order* (the ascending chains of lams may have infinite length and no upper bound). A classical example is the model of the *recursive* method contract (of `Math.fact_nc`)

$$\mathtt{Math.fact\_nc}\ a[\,](\_)\ \{\mathtt{0} + \mathtt{Math!fact\_nc}\ b[\,](\_) \to \_ \centerdot (a,b)\}\ \_$$

In order to circumvent this issue and to get a decision on deadlock-freedom in a finite number of steps, we use another usual method: running the Knaster-Tarski technique up-to a *fixed approximant*, let us say n, and then resorting to a *saturation argument*. If the n-th approximant is not a fixpoint, then the $(n+1)$-th approximant is computed by *reusing the same group names used by the n-th approximant* (no additional group name is created anymore). Similarly for the $(n+2)$-th approximant till a fixpoint is reached (by straightforward cardinality arguments, the fixpoint does exist, in this case). This fixpoint is called *the saturated state*. For example, in the case of the above contract, the n-th approximant returns the single state lam $[\![(a_1,a_2),\cdots,(a_{n-1},a_n)]\!]$. If we saturate at this stage, the next approximant returns the saturated state $[\![(a_1,a_2),\cdots,(a_{n-1},a_n),(a_2,a_2)]\!]$. This state contains a circular dependency – the pair (a_2,a_2) – revealing a potential deadlock in the corresponding program. Actually, in this case, this circularity is a *false* *positive* that is introduced by the (over)approximation: the original code never manifests a deadlock.

A more detailed account of the algorithm follows (a simplified version of the algorithm may be found in [9], see also Section 6). The model of lams is a partial order with a bottom element, which is the single state lam with the emptyset relation. For every syntactic operation on contracts, in particular $+$ and $\fatsemi$, we define a *monotone operation* on the model (an operation is monotone if, whenever it is applied to arguments in the order relation, it returns values in the same order relation). The algorithm analyzing contracts computes an *abstract class table* that associates with every method a function from tuples of group names to *pairs of lams*. The need for using pairs of lams, let them be $\langle \mathcal{W}, \mathcal{W}' \rangle$, is illustrated by means of an example. Consider the contract $\mathtt{c} = \mathtt{C!m}\ b[\ ](\ ) \rightarrow \_ \centerdot (a, b)$. This contract adds the dependency pair (a, b) to the current state. If the method `m` of class `C` only performs a method invocation, let it be $\mathtt{D!n}\ b[\ ](\ ) \rightarrow \_$ (without any `get` or `await` synchronization), then the invocation $\mathtt{C!m}\ b[\ ](\ ) \rightarrow \_$ does not contribute to the current state with other pairs. However it is possible that $\mathtt{D!n}\ b[\ ](\ ) \rightarrow \_$ introduces dependency pairs that affect the *future states* and that have nothing to do with (a, b). The same arguments apply in the case where `D!n` is a set of states: future dependency pairs are added according to what prescribed by the model of `D!n`. The dichotomy between present and future states allows us to augment the precision of our (compositional) abstract semantics. We notice that this dichotomy is not needed anymore for the main function. In fact, letting $\langle \mathcal{W}_{main}, \mathcal{W}'_{main} \rangle$ be the corresponding model, it is equivalent to the (single) lam $\mathcal{W}_{main} \cup \mathcal{W}'_{main}$ – in this case, futures are simply additional states to the current ones.

Back to the abstract class table, it is computed starting from the first approximant, which associates the function $\lambda \widetilde{a_{\mathtt{C,m}}}.\langle \mathbb{0}, \mathbb{0} \rangle$ with every method `C.m`. The next approximant is computed by transforming every entry of the lam class table according to the corresponding contract. When the saturated state is reached, the lam of the main function $\{\,\overline{T\ x\ ;}\ \ s\,\}$ is computed. Let $\langle \mathcal{W}_{main}, \mathcal{W}'_{main} \rangle$ be such lam. The input program is then deadlock-free if for every $W \in \mathcal{W}_{main} \cup \mathcal{W}'_{main}$, $W^{\mathtt{get}}$ has no circularity, where $W^{\mathtt{get}}$ is defined below.

Definition 1. *Let W be a set of group name dependencies. The* `get`-closure *of W, noted* $W^{\mathtt{get}}$*, is the least set such that*

$$W \in W^{\mathtt{get}} \qquad \frac{(a,b) \in W^{\mathtt{get}} \quad (b,c) \in W^{\mathtt{get}}}{(a,c) \in W^{\mathtt{get}}} \qquad \frac{(a,b) \in W^{\mathtt{get}} \quad (b,c)^{\mathtt{w}} \in W^{\mathtt{get}}}{(a,c) \in W^{\mathtt{get}}}$$

A set W contains a circularity *if the* `get`*-closure of its dependencies has a pair* (a, a).

As an example, we compute the abstract class table of the class `Math` in Figure 2. The contracts of such methods have been discussed in Section 3. Our analysis algorithm returns

method	first approx.	second approx.	third approx.
`Math.fact_g`	$\lambda a.\langle \mathbb{0}, \mathbb{0} \rangle$	$\lambda a.\langle [(a,a)], \mathbb{0} \rangle$	$\lambda a.\langle [(a,a)], \mathbb{0} \rangle$
`Math.fact_ag`	$\lambda a.\langle \mathbb{0}, \mathbb{0} \rangle$	$\lambda a.\langle [(a,a)^{\mathtt{w}}], \mathbb{0} \rangle$	$\lambda a.\langle [(a,a)^{\mathtt{w}}], \mathbb{0} \rangle$
`Math.fact_nc`	$\lambda a.\langle \mathbb{0}, \mathbb{0} \rangle$	$\lambda a.\langle [(a,b)], \mathbb{0} \rangle$	$\lambda a.\langle [(a,c),(c,d)], \mathbb{0} \rangle$

The fixpoints for `Math.fact_g` and `Math.fact_ag` are found at the third iteration. According to the above definition of deadlock-freeness, `Math.fact_g` yields a deadlock, whilst `Math.fact_ag` is deadlock-free. As discussed before, there exists no fixpoint for `Math.fact_nc`. If we decide to stop at the third iteration and saturate, we get $\lambda a.\langle [(a,c),(c,c),(c,d)], \mathbb{0}\rangle$, which contains a circularity. As we said before, this circularity is a false positive.

Note that saturation might even start at the first approximant (where every method is $\lambda a.\langle \mathbb{0}, \mathbb{0}\rangle$). In this case, for `Math.fact_g` and `Math.fact_ag`, we get the same answer and the same pair of lams as the above third approximant. For `Math.fact_nc` we get $\lambda a.\langle [(a,b),(b,b)], \mathbb{0}\rangle$, which contains a circularity. In general, it is possible to augment precision by delaying saturation. Consider the following abstract class table:

$$\begin{array}{ll}
\mathsf{C.m}: & a[\,](b[\,],c[\,])\ \{\mathsf{C.n}\ b[\,](c[\,]) \to \_ \,⨟\, \mathsf{C.n}\ c[\,](b[\,]) \to \_\}\ \_ \\
\mathsf{C.n}: & a[\,](b[\,])\ \{\mathsf{C.p}\ w[\,](a[\,]) \to \_ \,⨟\, \mathsf{C.p}\ b[\,](w'[\,]) \to \_\}\ \_ \\
\mathsf{C.p}: & a[\,](b[\,])\ \{\mathsf{C.q}\ b[\,]() \to \_ \centerdot (a,b)\}\ \_ \\
\mathsf{C.q}: & a[\,](\,)\ \{\mathbb{0}\}\ \_
\end{array}$$

This class table saturates at the second approximant and uses the same names w and w' in the two invocations of `C.n` inside `C.m`. This will produce a false positive. Saturating at the third approximant, instead, produces a precise response (the program is deadlock-free). We observe that the above abstract class table has a fixpoint at the fourth iteration.

Our technique is correct. We in fact demonstrate the following result.

Theorem 2. *Let $\langle \mathcal{W}_{main}, \mathcal{W}'_{main}\rangle$ be the lams of the main function of a* `core ABS` *program computed with an abstract class table (saturated at the n-th approximant, for some n). If no state of $\mathcal{W} \cup \mathcal{W}'$ has a circularity then the program is deadlock-free.*

5 The SDA Tool and Its Application to the Case Study

`ABS` (and, therefore, `core ABS`) comes with a suite [24] that offers a compilation framework, a set of tools to analyze the code, an Eclipse IDE plugin and Emacs mode for the language. We extended this suite with an implementation of our static deadlock analysis tool (SDA tool), available at `http://cs.unibo.it/~laneve/deadlock`. The SDA tool is built upon the abstract syntax tree (AST) of the ABS type checker. We can therefore exploit the type information stored in every node of the tree. This simplifies the implementation of several contract inference rules. The SDA tool is structured in three modules.

1. *Contract and Constraint Generation*. This is performed in three steps: i) the tool first parses the classes of the program and generates a map between interfaces and classes, required for the contract inference of method calls; ii) then it parses again all classes of the program to generate the initial environment Γ that maps methods to the corresponding method signatures; and iii) it finally parses the AST and, at each node, it applies the contract inference rules.
2. *Constraint Solving* is done by a generic semi-unification solver implemented in Java, following the algorithm defined in [12]. The implementation of that solver is available at `http://proton.inrialpes.fr/~mlienhar/semi-unification`.

When the solver terminates (and no error is found), it produces a substitution that validates the input constraints. Applying this substitution to the generated contracts produces the abstract class table and the contract of the main statement of the program.

3. *Contract Analysis* uses dynamic structures to store states of every method contract (because states become larger and larger as the analysis progresses). At each iteration of the analysis, a number of fresh group names is created and the states are updated according to what is prescribed by the contract. A basic operation of the analyzer is the renaming, which is used when computing every approximant. At each iteration, the tool checks whether a fixpoint has been reached. Saturation starts when the number of iterations reaches a maximum value (that may be customized by the user). In this case, since the precision of the algorithm degrades, the tool signals that the answer may be imprecise.

5.1 Simple Experiments

The SDA tool has been tested on a number of medium-size programs written for benchmarking purposes by ABS programmers. The programs may be found on the tool website; some of them (those with suffix `Mod`) required modifications to remove recursive object structures. The following table reports our experiments: for every program we display the number of lines, whether the analysis has reported a deadlock (`D`) or not (✓), and the time required for the analysis. With regards to time, we only report the time required by the contract inference system and the contract analysis when they run on a QuadCore 2.4GHz and Gentoo (Kernel 3.4.9):

program	lines	result	time
`PeerToPeer`	185	✓	0.474 sec
`BoundedBuffer`	103	✓	0.353 sec
`PingPongMod`	61	✓	0.046 sec
`MultiPingPongMod`	88	`D`	0.109 sec

5.2 The Industrial Case Study

The Fredhopper Access Server (FAS) is a distributed concurrent object-oriented system that provides search and merchandising services to eCommerce companies. FAS consists of a set of live environments and a single staging environment. A live environment processes queries from client web applications via web services and aims to provide a constant query capacity to client-side web applications. A staging environment is responsible for receiving data updates, and distributing the resulting indices across all live environments according to the *Replication Protocol*. The *Replication Protocol* has a single *SyncServer* module and one *SyncClient* module for each live environment. In turn, the *SyncServer* determines the schedule of replications, as well as their content, while a *SyncClient* receives data and configuration updates according to the schedule.

The *SyncServer* communicates to *SyncClient*s by creating *Worker* objects, which serve as the interface to the server-side of the *Replication Protocol*. On the other hand, *SyncClient*s schedule and create *ClientJob* objects to handle communications to the client-side of the *Replication Protocol*. When transferring data between the staging and the live environments, it is critical that the data remains *immutable*. To enforce immutability, without interfering with read/write accesses to the staging environment's underlying file system, the *SyncServer* creates a *Snapshot* object that encapsulates a snapshot of the necessary part of the staging environment's file system, and periodically *refreshes* it against the file system. This guarantees immutability of data until their update is deemed safe. The *SyncServer* uses a *Coordinator* object to determine the safe state in which the *Snapshot* can be refreshed.

5.3 The Application of SDA to FAS

As the *Replication Protocol* is a program with multiple threads interacting concurrently, there are risks of deadlock. In order to be able to to apply the SDA tool to the case study, we first made few adaptations.

We modified the `core` ABS model such that each interface defined in the model is implemented by at most one class. In particular we have restricted the types of replication items supported by the `core` ABS model to one. This change is adequate for deadlock analysis as these implementations only perform synchronous method calls or function calls with no scheduling point (`await` statements). In total we removed two implementations of replication item types.

We also removed all circular object structures. For example, in order to keep track of the *number* of `ClientJob` objects active at any given time, the `SyncClient` object keeps a list of references to such objects. On the other hand, each `ClientJob` object keeps a reference to its `SyncClient` object such that it can notify the `SyncClient` at the end of a replication session. We remove `SyncClient`'s reference to `ClientJob` such that `SyncClient` only increments an integer counter when a `ClientJob` is created and decrements the counter when a `ClientJob` object finishes a replication session. In total we modified three circular object structures to be non recursive.

Finally, we have annotated every `await` statement on boolean guards with a reference to the object that would resolve the expression to `True`. For example, during the interaction between `ClientJob` and `ConnectionThread`, `ClientJob` asynchronously invokes method `command(ListSchedule)` on `ConnectionThread` to ask the `ConnectionThread` to send all replication schedules, and then waits with the statement `await schedules != EmptySet`, where field `schedules` is subsequently set by `ConnectionThread` to transfer replication schedules on to the `ClientJob` object via method `receiveSchedule(Schedules)` (see Section 2.2). In this case we add the annotation `[thread]`, where `thread` is a reference to the `ConnectionThread` object. We have annotated 13 such `await` statements in total.

After these adaptations, we were ready to run the SDA tool. We ran it with number of iterations 1 and, within 40 seconds, we got the answer

```
### LOCK INFORMATION RESULTED BY THE ANALYSIS ###
  Saturation: true
  Deadlock in Main: false
```

In order to test the performance of SDA, we have also run it with other iteration values (which are not necessary for the functional analysis, in this case). The following table summarizes the results of our experiments:

Replication Protocol	time
`Iteration 1`	39.783 sec
`Iteration 2`	60.582 sec
`Iteration 3`	341.10 sec

We conclude with a remark about performance. The constraint inference is pseudo-linear in most of the cases. The fixpoint algorithm is exponential in the number of identifiers in a program. This is the reason why, in the above table, increasing the number of iterations (from 2 to 3) causes the runtime to increase by a factor of 6. We remark that in most cases, the precision of the SDA tool does not enhance at iterations higher than 1.

6 Related Works

A preliminary theoretical study was undertaken in [9], where (*i*) the considered language is a functional subset of `core ABS`; (*ii*) contracts are not inferred, they are provided by the programmer and type-checked; (*iii*) the deadlock analysis is less precise because it is not iterated as in this contribution, but stops at the first approximant, and (*iv*), more importantly, models of methods are not pairs of lams, which led it to discard dependencies (thereby causing the analysis, in some cases, to yield false negatives).

The proposals in the literature that statically analyze deadlocks are largely based on types. In [1, 3, 7, 22] a type system is defined that computes a partial order of the locks in a program and a subject reduction theorem demonstrates that tasks follow this order. On the contrary, our technique does not compute any ordering of locks, thus being more flexible: a computation may acquire two locks in different order at different stages, being correct in our case, but incorrect with the other techniques. In [17,20,21], Kobayashi and his colleagues use a very powerful technique, since they do not commit to any predefined partial order of locks and apply to codes with dynamic structures. However their concurrency models are different from that of `ABS` and a precise comparison is a matter for future work. Type-based deadlock analysis has also been studied in [18]. In this contribution, types define objects' states and can express acceptability of messages. The exchange of messages modifies the state of the objects. In this context, a deadlock is avoided by setting an ordering on types. With respect to our technique, [18] uses a deadlock prevention approach, rather than detection, and no inference system for types is provided. A number of model-theoretic techniques for deadlock analysis have also been defined.To mention one contribution (another one is [6], see below), in [4], circular dependencies among processes are detected as erroneous configurations, but dynamic creation of names is not treated.

Works that specifically tackle the problem of deadlocks for languages with the same concurrency model as that of `core ABS` are the following: [23] defines an approach for deadlock prevention (as opposed to our deadlock detection) in SCOOP, an Eiffel-based concurrent language. Different from our approach, they annotate classes with the used *processors* (the analogue of groups in ABS), while this information is inferred by our technique. Moreover each method exposes preconditions representing required lock ordering of processors (processors obeys an order in which to take locks), and this information must be provided by the programmer. [6] studies a Petri net based analysis, reducing deadlock detection to a reachability problem in Petri nets. This technique is more precise in that it is thread based and not just object based. Since the model is finite, this contribution does not address the feature of object creation and it is not clear how to scale the technique. We plan to extend our analysis in order to consider finer-grained thread dependencies instead of just object dependencies. [15] offers a design pattern methodology for CoJava to obtain deadlock-free programs. CoJava, a Java dialect where data-races and data-based deadlocks are avoided by the type system, prevents threads from sharing mutable data. Deadlocks are excluded by a programming style based on ownership types and *promise* (i.e. future) objects. The main differences with our technique are (*i*) the needed information must be provided by the programmer, (*ii*) deadlock freedom is obtained through ordering and timeouts, and (*iii*) no guarantee of deadlock freedom is provided by the system.

The work by Flores-Montoya *et al.* [8] and the corresponding DECO prototype deserve a separate discussion. They perform deadlock analysis on (a subset of) `core ABS` with a point-to analysis technique that returns a dependency graph. Then, in a clever way (by means of a may-happen-in-parallel analysis), unfeasible cycles in the dependency graph are discarded. The technique relies on an abstract evaluation of the code; therefore no inference system for extracting relevant informations is used. For this reason, the DECO tool does not manifest limitations of the current version of SDA, such as recursive object structures. As regards performance, DECO and SDA are comparable on small/mid-size programs (codes in Section 5.1). In case of the FAS module, DECO provides an answer in a bit more than 4 seconds. As regards the design, DECO is a monolithic code written in Prolog. On the contrary, SDA is a highly modular Java code (see Section 5). Every module may be replaced by another; for instance one may rewrite the inference system for another language and plug it easily in the tool, or one may use a different/refined contract analysis algorithm (see Conclusions).

7 Conclusions

We have developed a technique for statically detecting deadlocks in `core ABS` and discussed an industrial case study. The technique uses (i) an inference algorithm to extract abstract descriptions of methods, called contracts, and (ii) an evaluator of contracts, which computes an over-approximated fixpoint semantics.

This study can be extended in several directions. As regards the prototype, in the next release, we intend to remove most of the restrictions, as discussed in Section 2.2, since they have been considered only to ease the initial version. The next release of SDA will also provide indications about *how* deadlocks have been produced by pointing out the elements in the code that generated the detected circular dependencies. This way, the programmer will be able to check whether or not the detected circularities are actual deadlocks, fix the problem in case it is a verified deadlock, or be assured that his program is deadlock-free.

The current SDA tool is also able to capture (a form of) *livelock*, namely when several processes are continuously releasing and acquiring a set of group locks in a circular way. However, the theoretical development of this issue is at an early stage and we will extend the tool when the theory becomes more stable. SDA, being modular, may be integrated with other analysis techniques. In particular, we are prototyping the technique discussed in [10], which extends the theory of permutations to the contracts discussed in this paper. This technique provides a deadlock analysis that is complementary to the one discussed here. In the sense that there are programs that are false-positive in one technique and deadlock-free in the other, and conversely. Once this work is carried out, we will have an SDA tool with augmented precision.

Acknowledgment. We wish to thank Stephan Schroevers for proof reading this paper and the anonymous referees for useful suggestions and comments.

References

1. Abadi, M., Flanagan, C., Freund, S.N.: Types for safe locking: Static race detection for Java. ACM Trans. Program. Lang. Syst. 28 (2006)
2. The ABS Language Specification, ABS version 1.2.0 edition (September 2012), `http://tools.hats-project.eu/download/absrefmanual.pdf`
3. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe program.: preventing data races and deadlocks. In: Proc. OOPSLA 2002, pp. 211–230. ACM (2002)
4. Carlsson, R., Millroth, H.: On cyclic process dependencies and the verification of absence of deadlocks in reactive systems (1997)
5. Coppo, M.: Type inference with recursive type equations. In: Honsell, F., Miculan, M. (eds.) FOSSACS 2001. LNCS, vol. 2030, pp. 184–198. Springer, Heidelberg (2001)
6. de Boer, F., Bravetti, M., Grabe, I., Lee, M., Steffen, M., Zavattaro, G.: A petri net based analysis of deadlocks for active objects and futures. In: Păsăreanu, C.S., Salaün, G. (eds.) FACS 2012. LNCS, vol. 7684, pp. 110–127. Springer, Heidelberg (2013)
7. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: In PLDI 03: Programming Language Design and Implementation, pp. 338–349. ACM (2003)
8. Flores-Montoya, A.E., Albert, E., Genaim, S.: May-happen-in-parallel based deadlock analysis for concurrent objects. In: Beyer, D., Boteale, M. (eds.) FMOODS/FORTE 2013. LNCS, vol. 7892, pp. 273–288. Springer, Heidelberg (2013)
9. Giachino, E., Laneve, C.: Analysis of deadlocks in object groups. In: Bruni, R., Dingel, J. (eds.) FORTE/FMOODS 2011. LNCS, vol. 6722, pp. 168–182. Springer, Heidelberg (2011)
10. Giachino, E., Laneve, C.: A beginner's guide to the deadLock Analysis Model. In: TGC, Springer, Heidelberg (2013)
11. Giachino, E., Lascu, T.A.: Lock Analysis for an Asynchronous Object Calculus. In: Proc. 13th ICTCS (2012)
12. Henglein, F.: Type inference with polymorphic recursion. ACM Trans. Program. Lang. Syst. 15(2), 253–289 (1993)
13. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2011. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
14. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and System Modeling 6(1), 35–58 (2007)
15. Kerfoot, E., McKeever, S., Torshizi, F.: Deadlock freedom through object ownership. In: Wrigstad, T. (ed.) 5th International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO), in Conjunction with ECOOP 2009 (2009)

16. Kfoury, A.J., Tiuryn, J., Urzyczyn, P.: The undecidability of the semi-unification problem. Inf. Comput. 102(1), 83–101 (1993)
17. Kobayashi, N.: A new type system for deadlock-free processes. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 233–247. Springer, Heidelberg (2006)
18. Puntigam, F., Peter, C.: Types for active objects with static deadlock prevention. Fundam. Inform. 48(4), 315–341 (2001)
19. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing active objects to concurrent components. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 275–299. Springer, Heidelberg (2010)
20. Suenaga, K.: Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 155–170. Springer, Heidelberg (2008)
21. Suenaga, K., Kobayashi, N.: Type-based analysis of deadlock for a concurrent calculus with interrupts. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 490–504. Springer, Heidelberg (2007)
22. Vasconcelos, V.T., Martins, F., Cogumbreiro, T.: Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In: Proc. PLACES 2009. EPTCS, vol. 17, pp. 95–109 (2009)
23. West, S., Nanz, S., Meyer, B.: A modular scheme for deadlock prevention in an object-oriented programming model. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 597–612. Springer, Heidelberg (2010)
24. Wong, P.Y.H., Albert, E., Muschevici, R., Proença, J., Schäfer, J., Schlatte, R.: The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. Journal on Software Tools for Technology Transfer 14(5), 567–588 (2012)

Broadcast, Denial-of-Service, and Secure Communication

Roberto Vigo, Flemming Nielson, and Hanne Riis Nielson

Department of Applied Mathematics and Computer Science
Technical University of Denmark
{rvig,fnie,hrni}@dtu.dk

Abstract. A main challenge in the design of wireless-based Cyber-Physical Systems consists in balancing the need for security and the effect of broadcast communication with the limited capabilities and reliability of sensor nodes. We present a calculus of broadcasting processes that enables to reason about unsolicited messages and lacking of expected communication. Moreover, standard cryptographic mechanisms can be implemented in the calculus via term rewriting. The modelling framework is complemented by an executable specification of the semantics of the calculus in Maude, thereby facilitating solving a number of simple reachability problems.

Keywords: Cyber-Physical Systems, Broadcast communication, Denial-of-Service, Process calculus, Security protocol verification.

1 Introduction

Cyber-Physical Systems (CPSs) [1] are nowadays increasingly exploited in the realisation of critical infrastructure (e.g. power grid, healthcare, traffic control, defence). These are networks of sensors and actuators that monitor and interact with physical processes, often communicating on a wireless medium.

In a great many cyber-physical applications, the need for broadcast communication and some form of security conflicts with the limited capabilities of the sensors that are at the heart of these systems. On the one hand, broadcast (wireless) communication is often unavoidable, due to its reduced cost and ease of deployment, while cryptography is needed to ensure privacy, confidentiality, authentication, and other security properties. On the other hand, sensor nodes have limited computational capabilities and are typically powered by batteries, hence demanding for a careful use of the transceiver and of the on-board processor. In addition to this, sensors are often deployed in locations that lack of physical security, where environmental conditions and the presence of malicious parties are potential source of communication failure [2].

Coping with the coexistence of broadcast communication, security demands, limited computational power, and communication failure is therefore a main challenge in the design of reliable CPSs. Modelling frameworks and verification techniques are needed that can facilitate the work of the designers and developers of such systems

In this work we present a calculus of broadcasting processes, the Applied Quality Calculus, instrumented with a theory that allows modelling and reasoning about cryptographic primitives, and equipped with explicit notions of communication failure and unwanted communication. The calculus is complemented by an executable specification of its semantics in Maude, resulting in a simulation engine that can directly be used for prototyping and for solving bounded reachability problems.

The Applied Quality Calculus extends the Quality Calculus [3], and thus inherits the capability of reasoning about absence of expected communication and planning for default behaviour. In addition to this, the calculus is based on an asynchronous instant communication model, where a process is always allowed to perform an output (broadcast) and continue, while an input is allowed only when a matching output is being performed. This communication model faithfully represents wireless-based CPSs. Moreover, inputs are parametrised as to accept only messages with specific properties (e.g. format), ignoring unwanted communication and thus cutting down the confusion generated by broadcasting over a few (often just one) wireless channels.

Equational reasoning is implemented in the calculus by means of term rewriting, and leveraged for modelling both selective inputs and cryptographic operations. A rewrite theory for cryptographic primitives is presented, which relies on a simple yet powerful approach for defining cryptographic material, closer to real cryptosystems than other signature-based calculi, without the burden of an explicit type system.

Finally, the modelling expressiveness of the framework is illustrated on a meaningful case study, where two nodes of a wireless sensor network perform a key update exploiting asymmetric cryptography and secret sharing in order to hamper the work of an attacker. The example highlights how the calculus facilitates dealing with denial-of-service (expected information is not received), flooding generated by broadcast communication (receiving messages non-pertinent to the protocol), and cryptographic reasoning at the same time, and how the Maude implementation can support the verification of simple properties.

Related Work. The Calculus of Broadcasting Systems (CBS) [4] is the ancestor of a number of modern broadcasting calculi. In the subsequent studies, two main strands have flourished: on the one hand, theories for node mobility and dynamic topologies have been investigated; on the other hand, a number of calculi have been proposed that deal with low level characteristic of broadcast communication, such as transmission interference and range.

The calculus presented in [5] extends CBS with the notion of topology and presents an analysis for checking its consistency, but does not discuss the representation of cryptographic primitives in detail. Mobility in ad hoc networks is considered in [6], where a labelled characterisation of reduction barbed congruence is proposed. In [7], the dynamics of the topology is implicitly modelled in the semantics of the calculus. Different mobility models are studied in [8], together with their relationship to real applications.

The Calculus of Wireless Systems (CWS) [9] gives a lower-level representation of communication, modelling transmission interference in the semantics. CWS is extended in [10], where a timed scenario is considered which allows to study communication collisions and CSMA protocols. Interference is also considered in [11], where a notion of interference-sensitive preorder is introduced for mobile ad hoc networks.

Node mobility is studied together with limited transmission range in [12], combining the two lines of research mentioned above. A similar perspective is adopted in [13], which proposes a calculus where node mobility impacts on the reliability of transmissions in a probabilistic fashion. Finally, the calculus of [14] investigates different abstraction levels for describing dynamic networks, and considers the possibility of broadcasting at multiple transmission ranges.

Except for [5], none of the calculi mentioned so far provide any kind of equational reasoning on the messages that communicating parties exchange. In [15] a calculus for mobile ad hoc networks (CMAN) is devised, encompassing node mobility, spatially-oriented broadcast, and an implementation of cryptographic primitives via equations à la applied π-calculus [16]. Recently, [17] proposed a timed process calculus with fixed transmission ranges, where equational reasoning is parametrised on an inference system (our approach based on rewrite theories is similar, but allows a more expressive treatment of cryptographic primitives). Moreover, the calculus is equipped with a simulation theory, and the authors envision a possible mechanisation via Isabelle/HOL or Coq, in the spirit of our Maude implementation.

As for the analysis of reachability properties in wireless settings, [18] presents a broadcast version of the psi-calculi framework with an application to a routing protocol for mobile ad hoc networks.

Selective inputs have been proposed in [19] in connection to a generic pattern matching mechanism for cryptographic reasoning. Finally, [20] investigates in detail different ways of obtaining executable specification of structural operational semantics in rewriting logic, providing inspiring examples.

2 Syntax

The Applied Quality Calculus is a statically typed process calculus. According to the syntax displayed in Table 1, a process consists of actions that range over expressions. An expression e can be a variable x, a name a, or a function application $f(e_1, \dots, e_n)$. The syntax assumes to have a sorted signature Σ containing elements of the form $(f : s_f)$, where f is a function symbol and $s_f = t_1 \times \cdots \times t_n \to t$ is its sort, defining f as a function of arity n, the types $t_1, \dots, t_n$ of its arguments, and the type t of the expression computed by f. Constants are represented as functions of arity 0.

Data and Optional Data. The Applied Quality Calculus insists on the distinction between *data* and *optional data* introduced by the Quality Calculus and borrowed from programming languages like Standard ML. The reserved constant none is used to denote expressions which do not carry information, while the reserved function symbol $\mathsf{some}(\cdot)$ is used to denote expressions which evaluate to actual data. This distinction is formalised by means of a simple type system: the type D identifies expressions which convey information (data), whereas the type $D^?$ identifies expressions which possibly do not carry information (optional data). A name a has type D, a function f is typed according to its sort, none has type $D^?$, while $\mathsf{some}\colon D \to D^?$ takes an expression of type D and returns an expression of type $D^?$. A variable x could have either type D or $D^?$; in the following we will write x for variables of type $D^?$ and y to denote variables of type D.

Equational Reasoning. The behaviour of a function application is defined by a set $\mathcal{F}$ of conditional rewrite rules of the form

$$f(e_1, \ldots, e_n) \to e \text{ if } \mathit{cond}$$

where f and all the function symbols occurring in the e_i's belong to Σ, the e_i's contain all the variables appearing in e, and the side condition specifies some constraints on the e_i's. Valid constraints are limited to testing whether or not a list of names is in a given relation, e.g. whether or not two names form an asymmetric key pair, or checking whether a parameter e evaluates to an expression e', according to the following syntax:

$$\mathit{cond} ::= \overrightarrow{a} \in \mathcal{R} \,|\, e \vartriangleright e' \,|\, \mathit{cond} \wedge \mathit{cond} \,|\, \mathit{cond} \vee \mathit{cond} \,|\, \neg \mathit{cond} \,|\, \exists x.\mathit{cond}$$

where x has either type D or $D^?$ in $\exists x.\mathit{cond}$. The keyword $\mathsf{otherwise}$ is used in place of if *cond* to denote a rule which applies when no other rule can be applied for the given function symbol. We assume that e and the e_i's are typed coherently with the sort of the function f. Finally, we require that the rewrite system specified by $\mathcal{F}$ is confluent and terminating [21].

The restriction operator $(\nu \overrightarrow{a}; W)P$ declares the names $\overrightarrow{a} = (a_1, \ldots, a_m)$ as fresh in P and states a set of beliefs W on them. A belief $w \in W$ has the form $(a_i, \ldots, a_{i+k}) \in \mathcal{R}$, asserting that the tuple $(a_i, \ldots, a_{i+k})$ is in the relation $\mathcal{R}$. Given a restriction $(\nu \overrightarrow{a}; W)$, we require that $\mathsf{fn}(W) \subseteq \mathsf{fn}(\overrightarrow{a})$. In Sect. 2.1 relations will be introduced that contain symmetric keys (unary) and asymmetric key pairs (binary). In the following, W will denote the set of beliefs stated in a system so far, and we will call it *world*. A regular restriction is obtained specifying no belief on the restricted term, and it will be denoted $(\nu \overrightarrow{a})$.

The world W plays a key role in evaluating function applications, as the side condition of a rewrite rule can test whether or not some parameters are in a relation $\mathcal{R}$. Such a condition holds if the given relation is in W, as required by the semantics of Sect. 3. It is worth noting that a function application may have different evaluations in different worlds.

Processes and Quality Binders. As for the remaining operators, $e_1!e_2$ represents an asynchronous output of an expression e_2 on channel e_1. The input $e_1?x[e_2]$ waits for a message on channel e_1 and binds it to variable x if the expression

Table 1. Syntax of the Applied Quality Calculus

$$\begin{aligned}
P &::= \mathsf{0} \mid (\nu\overrightarrow{a}; W)P \mid e_1!e_2.P \mid b.P \mid P_1|P_2 \mid A(e) \\
&\quad\mid \mathsf{case}\ e\ \mathsf{of}\ \mathsf{some}(y)\colon P_1\ \mathsf{else}\ P_2 \\
b &::= e_1?x[e_2] \mid \&_{f(e_1,\ldots,e_m)}(b_1,\ldots,b_n) \\
e &::= x \mid a \mid f(e_1,\ldots,e_n) \mid \mathsf{none} \mid \mathsf{some}(e)
\end{aligned}$$

e_2 evaluates to $\mathsf{some}(c)$. When e_2 is a constant other than none we obtain the standard input operator $e?x$; when e_2 contains x we obtain an input operator able to select messages with specific properties: $c?x[\mathsf{fst}(x)]$, for example, accepts only pairs whose first component is not none. This is a very useful feature in a broadcast calculus, in particular when modelling system communicating over a single channel, as we will see in Sect. 4. $P_1|P_2$ is the parallel composition of two processes, and $A(e)$ is a call to a process defined in the system, with e being the actual parameter.

Finally, a quality binder $\&_q(e_1?x_1[e'_1],\ldots,e_n?x_n[e'_n])$ is used when n inputs are simultaneously active, and it is consumed when the quality guard q evaluates to true (tt). The quality guard q is a place-holder for a function application $f(e_1,\ldots,e_m)$, that states a condition to be met before proceeding with the computation (e.g. how many/which inputs must be performed), as explained in Sect. 2.1. The quality binder entails the distinction between data D and optional data $D^?$: when a binder $\&_q(e_1?x_1[e'_1],\ldots,e_n?x_n[e'_n])$ is passed, indeed, some inputs might have not received a value, if this is allowed by the guard q. Therefore, in the remainder of the computation, we need to record which inputs have been performed and which have not. The semantics achieves this goal by explicitly binding the variable of a non-performed input to none, and by binding the variable of a successful input to the constant expression $\mathsf{some}(c)$ that has been received.

Well-Formedness. As for typing, we require that

- in $e_1!e_2$ the channel e_1 and the outputted value e_2 have type D;
- in $e_1?x[e_2]$ the channel e_1 has type D, the input expression (or condition) e_2 has type $D^?$, and the input variable x has type $D^?$;
- in the call $A(e)$ the expression e has type $D^?$;
- in $\mathsf{case}\ e\ \mathsf{of}\ \mathsf{some}(y) : P_1\ \mathsf{else}\ P_2$ the expression e has type $D^?$, and the variable y has type D.

In the following we will write c and d to denote constant expressions of type D and $D^?$, respectively.

As the syntax is overly liberal in a number of respects, some restrictions help design well-formed processes. First, we will assume that expressions and processes are well-typed and that processes are closed (neither free variables nor free names are allowed). Secondly, we assume that input expressions and quality guards contain only variables that have been defined prior to the binder in which they occur. In particular, the process $\&_q(e_1?x_1, e_2?x_2[f(x_1)])$ is not well-formed, as the input on e_2 may arrive before the input on e_1, and in this case we would not be able to evaluate the input condition $f(x_1)$. Finally, limitations apply also to quality guards, as discussed in the following section.

2.1 Rewrite Rules for Cryptography and Quality Guards

Real cryptosystems lay down precise conditions that keys have to fulfil. In the Advanced Encryption Standard (AES), for example, a valid key must have a predefined length. The Applied Quality Calculus supports these limitations introducing explicit relations that apply to names, to be exploited in side conditions of rewrite rules. This is achieved without overly complicating the type system of the calculus.

Two simple relations are used to state that a name is a key:

- $a_1 \bowtie a_2$, meaning that $(a_1, a_2) \in \bowtie$ is a pair of keys in an asymmetric cryptosystem; we assume that a_1 is the private key and a_2 is the corresponding public key;
- $a\ltimes$, meaning that a is a key in a symmetric cryptosystem.

For instance, the process $(\nu a_1, a_2; a_1 \bowtie a_2)P$ declares a_1, a_2 as a new key pair in P, and in the trailing process every function application will be evaluated in the world $W = \{a_1 \bowtie a_2\}$. On the basis of these relations, a theory for cryptographic primitives is displayed in Table 2, in the wake of [22], that pioneered the rewriting approach to the symbolic modelling of cryptographic primitives (in turn inspired by the applied π-calculus). The main novelty of our approach is the use of conditions in rewrites for identifying keys.

Let us consider asymmetric cryptography as an illustrative example. Encryption and decryption are represented by the binary function symbols aenc and adec in Σ, respectively, while two rewrite rules in $\mathcal{F}$ are used to model the behaviour of asymmetric decryption. The side condition of the first rule requires that the keys y_2 (public) and y_3 (private) come from a valid key pair, i.e. are in the relation defined by $\bowtie$. If the condition does not hold or the first parameter is not an asymmetric encryption, then the decryption fails and none is returned. Intuitively, the second rule is applied only if the first rule cannot be applied. The formal treatment of $\mathsf{otherwise}$ relies on a transformation that translates a theory containing this keyword into a semantically equivalent theory without this attribute, as explained in [23, Sec. 4.5.4].

Quality Guards. Quality guards decide when a quality binder is satisfied and the trailing process can be executed. Let $\mathcal{B}$ be a subtype of D representing booleans, where the truth values $\{\mathsf{ff}, \mathsf{tt}\}$ are constants defined in the signature Σ. A quality guard $q(e_1, \ldots, e_m)$ for a binder $\&_{q(e_1,\ldots,e_m)}(b_1, \ldots, b_n)$ is a function $(q : t_1 \times \cdots \times t_m \times \mathcal{B}^n \rightarrow \mathcal{B}) \in \Sigma$ which takes as parameters

- m expressions $e_1, \ldots, e_m$ with types $t_1, \ldots, t_m$ (either D or $D^?$),
- n boolean parameters, each one stating whether or not a b_i has been satisfied,

Table 2. Function symbols and rules modelling cryptographic primitives and pairing

Σ	$\mathcal{F}$
$\mathsf{enc}: D \times D \to D$	$\mathsf{dec}(\mathsf{enc}(y_1, y_2), y_2) \to \mathsf{some}(y_1)$ if $y_2 \ltimes$
$\mathsf{dec}: D \times D \to D^?$	$\mathsf{dec}(y_1, y_2) \to$ none otherwise
$\mathsf{aenc}: D \times D \to D$	$\mathsf{adec}(\mathsf{aenc}(y_1, y_2), y_3) \to \mathsf{some}(y_1)$ if $y_3 \bowtie y_2$
$\mathsf{adec}: D \times D \to D^?$	$\mathsf{adec}(y_1, y_2) \to$ none otherwise
$\mathsf{sign}: D \times D \to D$	$\mathsf{getmessage}(\mathsf{sign}(y_1, y_2)) \to \mathsf{some}(y_1)$
$\mathsf{getmessage}: D \to D^?$	$\mathsf{getmessage}(y) \to$ none otherwise
$\mathsf{checksign}: D \times D \to D^?$	$\mathsf{checksign}(\mathsf{sign}(y_1, y_2), y_3) \to \mathsf{some}(y_1)$ if $y_2 \bowtie y_3$
	$\mathsf{checksign}(y_1, y_2) \to$ none otherwise
$\mathsf{hash}: D \to D$	
$\mathsf{pair}: D \times D \to D$	$\mathsf{fst}(\mathsf{pair}(y_1, y_2)) \to \mathsf{some}(y_1)$
$\mathsf{snd}: D \to D^?$	$\mathsf{fst}(y) \to$ none otherwise (likewise for snd)

and returns tt, in which case the binder is satisfied, or ff otherwise. A simple input evaluates to tt if it is performed, i.e. it transforms into a substitution, while it gives ff if at the time of the evaluation it has not been performed yet. As the n parameters related to the sub-binders are always there, we omit them for the sake of brevity.

In the Quality Calculus guards involve only the status of the sub-binders and are specified with predicates, denoted with $\forall$, $\exists$, $\exists!$, m/n, requiring to perform all the inputs, one input, exactly one input, or m out of n possible inputs before passing a binder, respectively. Rewrite rules can express predicates but allow to design also more elaborate guards. We can legally write, for example,

$$P \triangleq \&_{\exists}(c_1?x_1, c_2?x_2).\&_{q(x_1,x_2)}(c_1?x_3, c_2?x_4)$$

where the guard q is defined by the rule

$$q(x_1', x_2', y_1', y_2') \to (\mathsf{issome}(x_1') \wedge y_1') \vee (\mathsf{issome}(x_2') \wedge y_2')$$

The first two arguments of q are the input variables on which the first quality binder in P ranges, while the latter correspond to the boolean interpretation of the inputs of the second quality binder in P. The quality guard q states that the condition for consuming the second quality binder depends on the outcome of the first quality binder: if only the input concerning x_1 was performed, then x_3 must be bound to $\mathsf{some}(c)$, i.e. y_1 must be tt, and conversely x_4 must be $\mathsf{some}(c)$ if only the input involving x_2 was received.

The use of function $\mathsf{issome}(\cdot)$, checking if its argument is none, is crucial, as we can ask whether or not an input has been performed, but we should not inspect its content in a quality guard. Assume that a guard decides whether or not to pass a quality binder inspecting the content of received inputs: in this case we might end up in a situation in which all the inputs have been performed but the binder cannot be consumed due to what we received, and the process would be stuck since we have no means to rebind an input. Input expressions are entrusted of selecting inputs on the basis of their content, and such a test

Table 3. The structural congruence $\equiv$

$P \equiv P \qquad P_1 \equiv P_2 \Rightarrow P_2 \equiv P_1 \qquad P_1 \equiv P_2 \wedge P_2 \equiv P_3 \Rightarrow P_1 \equiv P_3 \qquad P\,\vert\,0 \equiv P$
$P_1\,\vert\,P_2 \equiv P_2\,\vert\,P_1 \qquad P_1\,\vert\,(P_2\,\vert\,P_3) \equiv (P_1\,\vert\,P_2)\,\vert\,P_3 \qquad A(e) \equiv P[e/x] \;\text{ if } A(x) \triangleq P$
$P_1 \equiv P_2 \Rightarrow C[P_1] \equiv C[P_2] \;\text{ where } C ::= [\,] \mid (\nu\overrightarrow{a};W)C \mid C\vert P \mid P\vert C$
$(\nu\overrightarrow{a};W)P \equiv P \;\text{ if } \overrightarrow{a} = (a_1,\ldots,a_m) \text{ and } \{a_1,\ldots,a_m\} \cap \mathsf{fn}(P) = \emptyset$
$(\nu\overrightarrow{a_1};W_1)(\nu\overrightarrow{a_2};W_2)P \equiv (\nu\overrightarrow{a_2};W_2)(\nu\overrightarrow{a_1};W_1)P$
$(\nu\overrightarrow{a};W)(P_1 \mid P_2) \equiv ((\nu\overrightarrow{a};W)P_1) \mid P_2 \;\text{ if } \overrightarrow{a} = (a_1,\ldots,a_m) \text{ and } \{a_1,\ldots,a_m\} \cap \mathsf{fn}(P_2) = \emptyset$

should be carefully avoided in quality guards. The example of Sect. 4 will show how to combine quality guards and input expressions.

3 Semantics

The semantics of the Applied Quality Calculus is defined by a structural equivalence $\equiv$ and a labelled transition relation $\overset{\alpha}{\Longrightarrow}$. The standard structural congruence is adapted to the new restriction operator, cf. Table 3. As usual, we assume to apply α-conversion wherever needed, in order to avoid accidental captures.

The semantics is parametrised on some auxiliary relations:

- $W \vdash e_1!e_2|b \rightarrow b'$, specifying the effect of the output $e_1!e_2$ on the binder b in the world W; technically, the syntax of binders has to be extended up consider substitutions of the form $[e/x]$;
- $b ::_v \theta$, for recording in $v \in \{\mathsf{ff}, \mathsf{tt}\}$ whether or not the binder b has been satisfied by the received inputs, that led to the substitutions recorded in θ. Observe that v has type $\mathcal{B}$, and thus can be passed as argument to a rewrite rule specifying a quality guard (boolean interpretation of binders);
- $W \vdash e \rhd e'$, describing how an expression e evaluates to a constant expression e' (either a c or a d) in the world W, according to the following rules:

$$\vdash a \rhd a \qquad \vdash \mathsf{none} \rhd \mathsf{none}$$

$$\frac{(f(e_1,\ldots,e_n) \rightarrow e \;\mathsf{if}\; \mathit{cond}) \in \mathcal{F} \qquad W \vdash \theta(\mathit{cond})}{W \vdash f(\theta(e_1'),\ldots,\theta(e_n')) \rightarrow \theta(e)}$$

$$\frac{W \vdash f(e_1,\ldots,e_n) \rightarrow^* u \quad \nexists u'.W \vdash u \rightarrow u'}{W \vdash f(e_1,\ldots,e_n) \rhd u}$$

A name a and the constant none always evaluate to themselves in every world. A world W supports a rewrite step if the side condition holds in W. A function application $f(e_1,\ldots,e_n)$ is evaluated applying rewriting steps until a non-reducible expression u is produced (i.e. a normal form). Observe that a rewrite step is viable only if there exists a rule $f(e_1',\ldots,e_n')$ if cond in $\mathcal{F}$ such that (i) the leftmost symbol corresponds to f (and arity and sort match), (ii) the formal parameters unify with the actual parameters under a most general unifier θ, (iii) the side condition (if any) holds in W after having undergone the substitutions in θ. If one of these conditions does not hold, then the evaluation fails and the process is stuck (the otherwise condition helps design robust function specification). $\rightarrow^*$ denotes the reflexive transitive closure of the rewriting relation $\rightarrow$, and $W \vdash t \rightarrow^* t'$ if W supports all the side conditions of at least one rewrite path which reduces t to t'. Since we require the rewrite system to be confluent and terminating, the evaluation strategy supporting the computation does not affect the result, even if it can heavily impact the performance.

We can now formally define $\mathsf{issome}\colon D^? \rightarrow \mathcal{B}$:

$$\mathsf{issome}(x) \rightarrow \mathsf{tt} \text{ if } x \rhd \mathsf{some}(c) \qquad\qquad \mathsf{issome}(x) \rightarrow \mathsf{ff} \text{ if } x \rhd \mathsf{none}$$

The expression evaluation relation allows also to define equality modulo rewriting, represented by the symbol $(=\colon D^{?^2} \rightarrow D^?) \in \Sigma$:

$$\begin{aligned} &=(x_1, x_2) \rightarrow \mathsf{some}(\mathsf{tt}) \text{ if } x_1 \rhd d \wedge x_2 \rhd d \\ &=(x_1, x_2) \rightarrow \mathsf{none} \qquad \mathsf{otherwise} \end{aligned}$$

In the following we will use the infix notation for the sake of simplicity.

The labelled transition relation $P \xRightarrow{\alpha} P'$ describes when a process P evolves into another process P', and it is instrumented with a label α which can be τ or an output action $c_1!c_2$. The transition relation is based on the relation $W \vdash P \xrightarrow[i]{\alpha} P'$, which enables $\xRightarrow{\alpha}$ under precise conditions. The relation $\xrightarrow[i]{\alpha}$ and its relationship with $\xRightarrow{\alpha}$ are defined in Table 4. $\xrightarrow[i]{\alpha}$ is labelled with an action α and an integer i, which can be either 0 (passive action) or 1 (active action). An active step $\xrightarrow[1]{\alpha}$ enables a transition $\xRightarrow{\alpha}$, as explained by the first rule, which also takes care of pushing restrictions to the top. The set W contains the beliefs accumulated with restrictions, and the second rule enables to evaluate rewrite conditions. As the world W affects expression and binder evaluation, it is a key component in determining the path followed by the computation: the possibility to execute a step $\xrightarrow[i]{\alpha}$ may depend on W. In order to highlight this relationship, we write $W \vdash P \xrightarrow[i]{\alpha} P'$.

The second group of clauses defines $\xrightarrow[i]{\alpha}$. Rule (Self) states that a process can silently evolve to itself, and it is central in enabling non-communicating processes to interleave. Rule (Brd) describes that performing an output is possibly an asynchronous active action (and thus can directly turn into a step $\xRightarrow{\alpha}$). The following two rules describe how an output affects a process guarded by a binder: if the output does not satisfy the binder (In-ff), then the related substitution is recorded and the binder modified accordingly; otherwise, if the output satisfies the binder (In-tt), the substitution computed so far is applied to the continuation process and the binder consumed. Observe that in both cases the action is passive, and therefore cannot directly enable a $\xRightarrow{\alpha}$ transition. This happens only when a synchronisation step takes place, as described by rule (Par). The composition of two processes that can make a transition $\xrightarrow[i]{\alpha}$ evolves only if the processes share the label α and the transitions are not both active. This implies that two input processes waiting for a value on the same channel can evolve together, as well as two processes that can synchronise: the synchronisation with an output process is actually the only case in which an input transition transforms into an active action, giving rise to a transition $\xRightarrow{\alpha}$. Thanks to this behaviour, the semantics realises broadcast communication: a binder accumulates expected outputs and the related substitutions, and when the synchronisation finally takes place with a matching output all the involved binders evolve at the same time. Observe that two outputs cannot evolve simultaneously, as they both are active actions. The last two rules of the group describe the evaluation of a case construct: the else branch is taken whenever the expression evaluates to none, otherwise the computation proceeds binding to the variable y the value to which the expression evaluates. Observe that a case transition is always labelled with τ, and thus cannot be mixed with input or output actions when composing two processes.

The clauses in the third group present how I/O substitutions are computed. An output affects an input binder only if they are performed on the same channel and the input condition evaluates to data when the outputted value is substituted to the input variable; otherwise it has no effect. Observe how broadcasting is realised within a single quality binder: sub-binders can be affected by the same output. This behaviour is coherent with the intended semantics of the quality binder, which states that n inputs are *simultaneously active* in $\&_q(e_1?x_1[e'_1], \ldots, e_n?x_n[e'_n])$, and therefore a number of them can synchronise with a single matching output.

The last group of clauses shows how a binder is evaluated, complementing the input clauses of the second group. In particular, a substitution evaluates to tt, since it is the result of a successful input, while a non-performed input evaluates to ff and maps the input variable to none. A quality binder is evaluated computing the function specified by the quality guard.

Finally, we require that a process is always in a stable configuration, according to which all the internal (silent) actions are performed before the possibility to synchronise with an external output vanishes. This implies that the process

$$c_1!c_2 \mid \mathsf{case}\ \mathsf{some}(c_1)\ \mathsf{of}\ \mathsf{some}(y){:}\ y?x.P\ \mathsf{else}\ Q$$

will always evolve to $P[\mathsf{some}(c_2)/x]$. Intuitively, by imposing this restriction we assume that internal actions are always processed faster than communicating actions (an on-board processor is faster than a transceiver).

4 Motivating Example

We demonstrate the flexibility of the Applied Quality Calculus modelling a hierarchical Wireless Sensor Network (WSN) where secret sharing to communicate security-critical information to a base station, as studied in [24].

Table 4. The labelled transition rules of the Applied Quality Calculus

$$\frac{P_1 \equiv (\nu \vec{a}; W)P_2 \quad W \vdash P_2 \xrightarrow[1]{\alpha} P_3 \quad (\nu \vec{a}; W)P_3 \equiv P_4}{P_1 \xRightarrow{\alpha} P_4} \qquad \frac{w \in W}{W \vdash w}$$

$$(\text{Self}) \ \vdash P \xrightarrow[0]{\tau} P \qquad (\text{Brd}) \ \frac{W \vdash e_1 \rhd c_1 \quad W \vdash e_2 \rhd c_2}{W \vdash e_1!e_2.P \xrightarrow[1]{c_1!c_2} P}$$

$$(\text{In-ff}) \ \frac{W \vdash c_1!c_2|b \to b' \quad b' ::_{\mathsf{ff}} \theta}{W \vdash b.P \xrightarrow[0]{c_1!c_2} b'.P} \qquad (\text{In-tt}) \ \frac{W \vdash c_1!c_2|b \to b' \quad b' ::_{\mathsf{tt}} \theta}{W \vdash b.P \xrightarrow[0]{c_1!c_2} P\theta}$$

$$(\text{Par}) \ \frac{W \vdash P_1 \xrightarrow[i]{\alpha} P_1' \quad W \vdash P_2 \xrightarrow[j]{\alpha} P_2'}{W \vdash P_1 \mid P_2 \xrightarrow[i+j]{\alpha} P_1' \mid P_2'} \quad i + j \leq 1$$

$$\frac{W \vdash e \rhd \mathsf{some}(c)}{W \vdash \mathsf{case}\ e\ \mathsf{of}\ \mathsf{some}(y): P_1\ \mathsf{else}\ P_2 \xrightarrow[1]{\tau} P_1[c/y]} \qquad \frac{W \vdash e \rhd \mathsf{none}}{W \vdash \mathsf{case}\ e\ \mathsf{of}\ \mathsf{some}(y): P_1\ \mathsf{else}\ P_2 \xrightarrow[1]{\tau} P_2}$$

$$\frac{W \vdash e_1 \rhd c_1 \quad W \vdash e_2[c_2/x] \rhd \mathsf{some}(c_3)}{W \vdash c_1!c_2|e_1?x[e_2] \to [\mathsf{some}(c_2)/x]} \qquad \frac{W \vdash e_1 \rhd c_1 \quad W \vdash e_2[c_2/x] \rhd \mathsf{none}}{W \vdash c_1!c_2|e_1?x[e_2] \to e_1?x[e_2]}$$

$$\frac{W \vdash e_1 \rhd c_1'}{W \vdash c_1!c_2|e_1?x[e_2] \to e_1?x[e_2]} \quad c_1' \neq c_1 \qquad \frac{W \vdash c_1!c_2|b_1 \to b_1' \cdots W \vdash c_1!c_2|b_n \to b_n'}{W \vdash c_1!c_2|\&_q(b_1, \ldots, b_n) \to \&_q(b_1', \ldots, b_n')}$$

$$e_1?x[e_2] ::_{\mathsf{ff}} [\mathsf{none}/x] \qquad [\mathsf{some}(c)/x] ::_{\mathsf{tt}} [\mathsf{some}(c)/x]$$

$$\frac{b_1 ::_{v_1} \theta_1 \quad \cdots \quad b_n ::_{v_n} \theta_n}{\&_{q(e_1,\ldots,e_m)}(b_1, \ldots, b_n) ::_v \theta_n \cdots \theta_1} \quad q(e_1, \ldots, e_m, v_1, \ldots, v_n) \rhd v$$

In such a scenario, each message is broadcast over a single wireless channel and can thus be eavesdropped by an attacker. A (k, m)-threshold-scheme can be applied to security-critical messages in order to hamper the work of the adversary, who is required to intercept at least k shares (or shadows) before obtaining a message (and trying to break an encryption scheme). Moreover, we assume that the communication may fail, due to environmental conditions, hardware failures, or the attacker's intervention, and we exploit quality binders in order to design processes robust against this sort of denial-of-service. Finally, the base station is periodically receiving data from sensor nodes in its range, which measure some physical parameters of the environment. Nonetheless, when updating the session key, the base station ignores messages sent by the sensors, in order to accomplish this critical task as quickly as possible. As all the communications take place on a single channel, we make use of input conditions to distinguish among messages.

Secret Sharing. A shadow is represented by a function $(\mathsf{share} : D^3 \to D) \in \Sigma$, the first parameter being the secret, the second the number of shares needed for reconstructing it, and the third an identifier which helps distinguish different shares. The reconstruction step is modelled with a function ($\mathsf{combine} : D^k \to D^?) \in \Sigma$, which takes k shares and returns the secret only if they are all different. For fixed $k = m = 3$, we obtain the following implementation:

$$\begin{array}{ll} \mathsf{combine}(\mathsf{share}(y_1, 3, y_1'), \mathsf{share}(y_1, 3, y_2'), \mathsf{share}(y_1, 3, y_3')) \to \mathsf{some}(y_1) & \\ & \text{if } y_i' \rhd c_i \wedge i \neq j \Rightarrow c_i \neq c_j \\ \mathsf{combine}(y_1, y_2, y_3) \to \mathsf{none} & \text{otherwise} \end{array}$$

Note that the c_i's are of type D. We omit the parameter m for simplicity, but it could be included to capture the fact that shadows of a same secret s belonging to different schemes cannot be used to rebuild s.

The Protocol. Consider now a WSN in which a central unit CU has to communicate a new symmetric session key to a base station under its control. CU generates a new symmetric key k, signs it with its secret key sk_{CU}, computes 3 shares, encrypts the shadows under the base station public key pk_{BS}, and finally communicates the shares on a wireless channel c after having notified the base station that an update transaction is starting:

$$\begin{array}{ll} \mathsf{CU} \triangleq & (\nu k; k \ltimes)\mathsf{c!start\_transaction} \\ & \mathsf{c!pair}(\mathsf{aenc}(\mathsf{share}(\mathsf{sign}(k, sk_{\mathsf{CU}}), 3, 1), pk_{\mathsf{BS}}), 1) \\ & \mathsf{c!pair}(\mathsf{aenc}(\mathsf{share}(\mathsf{sign}(k, sk_{\mathsf{CU}}), 3, 2), pk_{\mathsf{BS}}), 2) \\ & \mathsf{c!pair}(\mathsf{aenc}(\mathsf{share}(\mathsf{sign}(k, sk_{\mathsf{CU}}), 3, 3), pk_{\mathsf{BS}}), 3) \\ & \mathsf{set}_1!\mathsf{t}_1.\&_{\exists}(\mathsf{c}?x_e[\mathsf{dec}(x_e, k) = \mathsf{some}(\mathsf{end\_transaction})], \mathsf{tick}_1?x_{t1}) \\ & \mathsf{case}\ x_e\ \mathsf{of}\ \mathsf{some}(y_e) \colon \mathsf{set}_2!\mathsf{t}_2.\mathsf{tick}_2?x_{t2}.\mathsf{CU}\ \mathsf{else}\ \mathsf{CU} \end{array}$$

The process CU makes use of constants to represent integers (threshold-scheme parameters and share identifiers). After having issued the new key (signed and then split in three encrypted shadows), the central unit sets a local timer to t_1 and waits for a notification from the base station to arrive within the prescribed time. The time expires when a message is received from the timer on channel tick_1. Observe that the first input is instrumented with a condition that tests whether or not the received message corresponds to end_transaction encrypted under the new key k. At this point the central unit is enabled to check which input triggered passing the binder, and thus decide if the key update was successful or not: in the event the key has been updated, the central unit waits for t_2 unit of time and then starts a new update transaction, otherwise the new transaction is started immediately. A new timer is used to avoid message confusion. Observe that there is no need to decrypt x_e since its content has already been tested in the input condition: we only need to check that it is not none.

It is worth noting that the central unit is robust with respect to the event that the base station does not respond, thanks to the use of the existential quality guard and of the local clock, which always responds.

The base station is defined by a process BS, which waits for three shares and then sends an acknowledgement back to CU, encrypted under the new key.

```
$\mathsf{BS} \triangleq \mathsf{c}?x.\mathsf{case}\ x\ \mathsf{of\ some}(y_1):$
    $\mathsf{case}\ y_1 = \mathsf{end\_transaction\ of\ some}(y_2): \mathsf{set_3!t_3}$
      $\&_\exists(\&_\forall(\mathsf{c}?z_1[\mathsf{snd}(z_1) = \mathsf{some}(1)], \mathsf{c}?z_2[\mathsf{snd}(z_2) = \mathsf{some}(2)]$
            $\mathsf{c}?z_3[\mathsf{snd}(z_3) = \mathsf{some}(3)]),$
         $\mathsf{tick_3}?x_{t3})$ ... (extract the shares in $y'_1, y'_2, y'_3$) ...
            $\mathsf{case\ combine}(y'_1, y'_2, y'_3)\ \mathsf{of\ some}(y_s):$
              $\mathsf{case\ checksign}(y_s, pk_{\mathsf{CU}})\ \mathsf{of\ some}(y_k):$
                $\mathsf{c!enc}(\mathsf{end\_transaction}, y_k).\mathsf{BS\ else\ 0}$
              $\mathsf{else\ 0}$
            $\mathsf{else\ 0}$ ... (if share extraction fails then shut down) ...
    $\mathsf{else\ store!}y_1.\mathsf{BS}$
  $\mathsf{else\ 0}$
```

If BS receives a reading from a sensor, the value is stored in the base station memory (simulated by channel store). If an update instruction is received, then the base station waits for three shares. BS must wait for all the shadows to arrive, and thus uses a quality binder instrumented with the $\forall$ guard. Furthermore, in order to discard messages from the sensors, the inputs within the binder rely on a condition matching only a fixed pattern. For the sake of brevity, we have omitted a number of case constructs needed to compute projections and to decrypt the result. When three messages are received within time t_3, the base station tries to compute the original secret: if the operation succeeds then the signature is verified using the public key pk_{CU} of CU, and then an acknowledgement is sent back to the central unit.

Observe that the base station continues to behave in a planned manner even if the information expected from the central unit does not arrive. If the shares are not received within time t_3, their combination or the verification of the signature fail, then BS is automatically switched off for security reasons.

5 Executable Specification of the Semantics

We have developed an executable specification of the semantics of the calculus in Maude [23], in the wake of [20]. The rewriting theory $\mathcal{F}$ is encoded as a set of conditional equations, with which Maude can directly deal. As for the semantics, the central idea is to generate a rewrite rule of the form $P \to Q$ `if` $P_1 \to Q_1 \wedge \cdots \wedge P_n \to Q_n$ for each inference rule with premises $P_1 \to Q_1, \ldots, P_n \to Q_n$ and conclusion $P \to Q$, thus transforming transitions into rewrites. For the sake of simplicity and performance we encoded the semantics in a way that is directly executable by Maude's default interpreter, that is, variables in the conclusion of a rule (right-hand side) either appear in the premise or in the conditions (admissible module). This is not the case of the input rules of the semantics of Table 4, where we are implicitly quantifying over all possible outputs. In order to overcome this inconvenient, we introduced extended processes in which we record available broadcast values. Formally establishing the equivalence of the two semantics is part of our future work.

Besides the usefulness for prototyping and debugging, the possibility to simulate the execution of a system helps deal with reachability problems. Our experiments with the implementation exploit the powerful `search` command of Maude's default interpreter, by means of which bounded reachability problems can be simply solved. As regards the example of Sect. 4, we verified that whenever the base station receives the clock signal before the three shares, then it shuts down. Obviously, any infinite state system can only be simulated up to a certain bound, but bounded reachability can be an effective verification method if we consider that a node lifetime is limited in a WSN, and that also the attacker usually has limited resources.

6 Conclusion and Future Work

The characteristics of typical components of CPSs and the nature of the environment in which they are deployed demand for designing software that is both robust against lacking communication and able to ignore unwanted information. This is a complex task *per se*, and it is even harder in those applications that require some degree of security and need a broadcast communication model, where everyone hears everyone.

The framework we have presented facilitates the design of CPSs by providing a calculus that is naturally equipped with the notion of absence of communication and selective inputs. Denial-of-service is addressed by resorting to the distinction between data and optional data introduced by the Quality Calculus. A single mechanism based on rewrite rules is leveraged to implement both selective inputs and cryptographic reasoning, and it is also exploited to design elaborate quality guards, more expressive than propositional predicates. Moreover, a simple yet powerful approach to the definition of cryptographic material has been introduced. The expressiveness of the framework has been discussed on a meaningful example, and some results obtained via the implementation of the semantics of the calculus in Maude have been sketched.

Future work includes further investigation of verification techniques based on term rewriting. It seems promising to study abstraction approaches for analysing infinite state systems beyond bounded reachability, in the wake of studies carried out in Maude's community. Moreover, it would be interesting to consider a wider class of CPSs with component mobility, thus enriching the framework with a notion of network topology and spatially-bounded broadcast.

Acknowledgement. This work is supported by the IDEA4CPS project, granted by the Danish Research Foundations for Basic Research (DNRF86-10), and by MT-LAB, a VKR Centre of Excellence for the Modelling of Information Technology.

References

1. Lee, E.A.: Cyber Physical Systems: Design Challenges. In: Int. Symp. on Object/Component/Service-Oriented Real-Time Distributed Computing (2008)
2. Neuman, C.: Challenges in Security for Cyber-Physical Systems. In: DHS Workshop on Future Directions in Cyber-Physical Systems Security (2009)
3. Nielson, H.R., Nielson, F., Vigo, R.: A Calculus for Quality. In: Păsăreanu, C.S., Salaün, G. (eds.) FACS 2012. LNCS, vol. 7684, pp. 188–204. Springer, Heidelberg (2013)
4. Prasad, K.: A calculus of broadcasting systems. Science of Computer Programming 25(2-3), 285–327 (1995)
5. Nanz, S., Hankin, C.: A framework for security analysis of mobile wireless networks. Theoretical Computer Science 367(1-2), 203–227 (2006)
6. Merro, M.: An Observational Theory for Mobile Ad Hoc Networks. Inf. Comput. 207(2), 194–208 (2009)
7. Ghassemi, F., Fokkink, W., Movaghar, A.: Equational Reasoning on Mobile Ad Hoc Networks. Fundamenta Informaticae 105(4), 375–415 (2010)
8. Godskesen, J.C., Nanz, S.: Mobility Models and Behavioural Equivalence for Wireless Networks. In: Field, J., Vasconcelos, V.T. (eds.) COORDINATION 2009. LNCS, vol. 5521, pp. 106–122. Springer, Heidelberg (2009)
9. Lanese, I., Sangiorgi, D.: An operational semantics for a calculus for wireless systems. Theoretical Computer Science 411(19), 1928–1948 (2010)
10. Merro, M., Ballardin, F., Sibilio, E.: A timed calculus for wireless systems. Theoretical Computer Science 412(47), 6585–6611 (2011)
11. Bugliesi, M., Gallina, L., Marin, A., Rossi, S., Hamadou, S.: Interference-Sensitive Preorders for MANETs. In: 9th Int. Conf. on Quantitative Evaluation of Systems (QEST), pp. 189–198 (2012)
12. Singh, A., Ramakrishnan, C., Smolka, S.A.: A process calculus for Mobile Ad Hoc Networks. Science of Computer Programming 75(6), 440–469 (2010)
13. Song, L., Godskesen, J.C.: Probabilistic Mobility Models for Mobile and Wireless Networks. In: Theoretical Computer Science - 6th IFIP Int. Conf., pp. 86–100 (2010)
14. Kouzapas, D., Philippou, A.: A Process Calculus for Dynamic Networks. In: Bruni, R., Dingel, J. (eds.) FORTE 2011 and FMOODS 2011. LNCS, vol. 6722, pp. 213–227. Springer, Heidelberg (2011)
15. Godskesen, J.C.: A Calculus for Mobile Ad Hoc Networks. In: Murphy, A.L., Vitek, J. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 132–150. Springer, Heidelberg (2007)
16. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: ACM Symp. on Principles of Programming Languages (POPL), pp. 104–115 (2001)
17. Macedonio, D., Merro, M.: A Semantic Analysis of Wireless Network Security Protocols. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 403–417. Springer, Heidelberg (2012)
18. Borgström, J., Huang, S., Johansson, M., Raabjerg, P., Victor, B., Åman Pohjola, J., Parrow, J.: Broadcast Psi-calculi with an Application to Wireless Protocols. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 74–89. Springer, Heidelberg (2011)
19. Buchholtz, M., Nielson, H.R., Nielson, F.: A Calculus for Control Flow Analysis of Security Protocols. Int. J. of Information Security 2(3-4), 145–167 (2004)

20. Verdejo, A., Martí-Oliet, N.: Executable structural operational semantics in Maude. Journal of Logic and Algebraic Programming 67(1-2), 226–293 (2006)
21. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
22. Blanchet, B.: From Secrecy to Authenticity in Security Protocols. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, p. 342. Springer, Heidelberg (2002)
23. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude Manual, Version 2.6 (2011)
24. Shu, T., Liu, S., Krunz, M.: Secure Data Collection in Wireless Sensor Networks Using Randomized Dispersive Routes. IEEE Transactions on Mobile Computing 9(7), 941–954 (2010)

Characterizing Fault-Tolerant Systems by Means of Simulation Relations

Ramiro Demasi[1], Pablo F. Castro[2,3], Thomas S.E. Maibaum[1], and Nazareno Aguirre[2,3]

[1] Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada
demasira@mcmaster.ca, tom@maibaum.org

[2] Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Río Cuarto, Córdoba, Argentina
{pcastro,naguirre}@dc.exa.unrc.edu.ar

[3] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina

Abstract. In this paper, we study a formal characterization of fault-tolerant behaviors of systems via simulation relations. This formalization makes use of particular notions of simulation and bisimulation in order to compare the executions of a system that exhibit faults with executions where no faults occur. By employing variations of standard (bi)simulation algorithms, our characterization enables us to algorithmically check fault-tolerance, i.e., to verify that a system behaves in an acceptable way even under the occurrence of faults.

Our approach has the benefit of being simple and supporting an efficient automated treatment. We demonstrate the practical application of our formalization through some well-known case studies, which illustrate that the main ideas behind most fault-tolerance mechanisms are naturally captured in our setting.

1 Introduction

The increasing demand for highly dependable and constantly available systems has brought attention to providing strong guarantees for software correctness, especially for safety critical systems. Some examples of such critical systems include software for medical devices and software controllers in the avionics and automotive industries. In this context, a problem that deserves attention is that of capturing *faults*, understood as unexpected events that affect a system and may corrupt or degrade its performance, as well as expressing and reasoning about the properties of systems in the presence of faults.

The field of fault-tolerant systems is concerned with providing techniques that can be used to increase the fault-tolerance characteristics of software, or computer systems in general. This includes specific mechanisms for achieving fault-tolerance, as well as for appropriately modeling fault-tolerant systems, and expressing and reasoning about fault-tolerant behaviors. Some examples of traditional techniques employed to deal with fault-tolerance are: *component* *replication*, *N-version programming*, *exception mechanisms*, *transactions*, etc. Some emerging approaches try to deal with fault-tolerance in formal settings, with the aim of mathematically *proving* that a given system effectively tolerates faults. For example, in [9], an approach to design and verify programs that tolerate faults, where faults are formalized as operations performed at random time intervals, is proposed. Another example of formal approach to fault-tolerance is that presented in [3–5], where Unity programs are complemented with fault steps, and the logic underlying Unity is used to prove properties of programs. More recently, formal approaches involving model checking, applied to fault-tolerance, have been proposed. In these approaches, temporal logics are employed to capture fault-tolerance properties of reactive systems, and then model checking algorithms are used to automatically verify that these properties hold for a given system. Since model checking provides fully automated analysis, and counterexamples are generated when a property does not hold (which is extremely helpful in finding the source of the problem in the system), model checking based approaches to fault-tolerance provide significant benefits over other semi-automated or manual formal approaches. However, the languages employed for the description of systems and system properties in model checking do not provide a built-in way of distinguishing between normal and abnormal behaviors. Thus, when capturing fault-tolerant systems, and expressing fault-tolerance properties, the specifier needs to *encode* in some suitable way the faults and their consequences. This makes formulas longer and more difficult to understand, which has a negative impact on analysis, since the performance of model checking algorithms depends on the length of the formula being analyzed.

In this paper, we propose an alternative formal approach for dealing with the analysis of fault-tolerance, which allows for a fully automated analysis, and appropriately distinguishes faulty behaviors from normal ones. This approach provides a formalism for modeling fault-tolerant systems that features a built-in notion of abnormal transition, to capture faults. The notion of fault-tolerance is characterized by defining simulation/bisimulation relations, between the desired "fault-free" program, and that which tolerates faults. Since, as it is well known, a system may tolerate faults exhibiting different degrees of so called *fault-tolerance*, different simulation/bisimulation relations are provided for different kinds of fault-tolerance. More precisely, the kinds of fault-tolerance that we capture in our setting are *masking*, *nonmasking* and *failsafe*. *Masking fault-tolerance* corresponds to the case in which the system may completely tolerate the faults, not allowing these to have any observable consequences for the users; *nonmasking fault-tolerance* corresponds to the case in which, after a fault occurs, the system may undergo some process to eventually take the system back to a "good" behavior; finally, *failsafe fault-tolerance* corresponds to the case in which the system may react to a fault by switching to a behavior that is safe but in which the system is restricted in its capacity. Since in this approach fault-tolerance is captured via bisimulation, one is able to check that a system tolerates faults to some degree (masking, nonmasking, failsafe), without the need for user intervention, by employing (bi)simulation algorithms.

2 Preliminaries

In this section we introduce some concepts that will be necessary throughout the paper. For the sake of brevity, we assume some basic knowledge on model checking; the interested reader may consult [7]. We model fault-tolerant systems by means of *colored Kripke structures*, as introduced in [8]. Given a set of propositional letters $AP = \{p, q, s, \ldots\}$, a *colored Kripke structure* is a 5-tuple $\langle S, I, R, L, \mathcal{N}\rangle$, where S is a set of states, $I \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a transition relation, $L : S \to \wp(AP)$ is a labeling function indicating which propositions are true in each state, and $\mathcal{N} \subseteq S$ is a set of *normal*, or "green" states. The complement of $\mathcal{N}$ is the set of "red", abnormal or faulty states. Arcs leading to abnormal states (i.e., states not in $\mathcal{N}$) can be thought of as faulty transitions, or simply *faults*. Then, normal executions are those transiting only through green states. The set of normal executions is denoted by $\mathcal{NT}$. We assume that in every colored Kripke structure, and for every normal state, there exists at least one successor state that is also normal, and that at least one initial state is green. This guarantees that every system has at least one normal execution, i.e., that $\mathcal{NT} \neq \emptyset$.

As is usual in the definition of temporal operators, we employ the notion of trace. Given a colored Kripke structure $M = \langle S, I, R, L, \mathcal{N}\rangle$, a *trace* is a maximal sequence of states, whose consecutive pairs are adjacent wrt. R. When a trace of M starts in an initial state, it is called an *execution* of M, with *partial* executions corresponding to non-maximal sequences of adjacent states starting in an initial state. Given a trace $\sigma = s_0, s_1, s_2, s_3, \ldots$, the ith state of σ is denoted by $\sigma[i]$, and the final segment of σ starting in position i is denoted by $\sigma[i..]$. Moreover, we distinguish among the different kinds of outgoing transitions from a state. We denote by $\dashrightarrow$ the restriction of R to faulty transitions, and $\rightarrow$ the restriction of R to non-faulty transitions. We define $Post_N(s) = \{s \in S |\ s \rightarrow s'\}$ as the set of successors of s reachable via non-faulty (or good) transitions; similarly, $Post_F(s) = \{s \in S |\ s \dashrightarrow s'\}$ represents the set of successors of s reachable via faulty arcs. Analogously, we define $Pre_N(s')$ and $Pre_F(s')$ as the set of predecessor of s' via normal and faulty transitions, respectively. Moreover, $Post^*(s)$ denotes the states which are reachable from s. Without loss of generality, we assume that every state has a successor [7]. We denote by $\Rightarrow^*$ the transitive closure of $\dashrightarrow \cup \rightarrow$.

In order to state properties of systems, we use a fragment of dCTL [8], a branching time temporal logic with deontic operators designed for fault-tolerant system verification. Formulas in this fragment, that we call dCTL-, refer to properties of behaviors of colored Kripke structures, in which a distinction between *normal* and *abnormal* states (and therefore also a distinction between normal and abnormal traces) is made. The logic dCTL is defined over the Computation Tree Logic CTL, with its novel part being the deontic operators $\mathbf{O}(\psi)$ (obligation) and $\mathbf{P}(\psi)$ (permission), which are applied to a certain kind of path formula ψ. The intention of these operators is to capture the notion of *obligation* and *permission* over traces. Intuitively, these operators have the following meaning:

– $\mathbf{O}(\psi)$: property ψ is obliged in every future state, reachable via non-faulty transitions.
– $\mathbf{P}(\psi)$: there exists a normal execution, i.e., not involving faults, starting from the current state and along which ψ holds.

Obligation and permission will enable us to express intended properties which should hold in *all* normal behaviors and *some* normal behaviors, respectively. These deontic operators have an implicit *temporal* character, since ψ is a path formula. Let us present the syntax of dCTL-. Let $AP = \{p_0, p_1, \ldots\}$ be a set of atomic propositions. The sets Φ and Ψ of *state formulas* and *path formulas*, respectively, are mutually recursively defined as follows:

$$\begin{aligned}\Phi &::= p_i \mid \neg\Phi \mid \Phi \to \Phi \mid \mathsf{A}(\Psi) \mid \mathsf{E}(\Psi) \mid \mathbf{O}(\Psi) \mid \mathbf{P}(\Psi)\\ \Psi &::= \mathsf{X}\Phi \mid \Phi\, \mathcal{U}\, \Phi \mid \Phi\, \mathcal{W}\, \Phi\end{aligned}$$

Other boolean connectives (here, state operators), such as $\wedge$, $\vee$, etc., are defined as usual. Also, traditional temporal operators G and F can be expressed, as $\mathsf{G}(\phi) \equiv \phi\ \mathcal{W}\ \bot$, and $\mathsf{F}(\phi) \equiv \top\ \mathcal{U}\ \phi$. The standard boolean operators and the CTL quantifiers A and E have the usual semantics. Now, we formally state the semantics of the logic. We start by defining the relation $\vDash$, formalizing the satisfaction of dCTL- state formulas in colored Kripke structures. For the deontic operators, the definition of $\vDash$ is as follows:

– $M, s \vDash \mathbf{O}(\psi) \Leftrightarrow$ for every $\sigma \in \mathcal{NT}$ such that $\sigma[0] = s$ we have that for every $i \geq 0$ $M, \sigma[i..] \vDash \psi$.
– $M, s \vDash \mathbf{P}(\psi) \Leftrightarrow$ for some $\sigma \in \mathcal{NT}$ such that $\sigma[0] = s$ we have that for every $i \geq 0$ $M, \sigma[i..] \vDash \psi$.

For the standard CTL operators the definition of $\vDash$ is as usual (see [7]).

We denote by $M \vDash \varphi$ the fact that $M, s \vDash \varphi$ holds for every state s of M, and by $\vDash \varphi$ the fact that $M \vDash \varphi$ for every colored Kripke structure M.

In order to illustrate the semantics of the deontic operators, let us consider the colored Kripke structure in Figure 1, where the set of propositional variables is $\{p, q, r, t\}$, and each state is labeled by the set of propositional variables that hold in it. The states that are the target of dashed arcs are abnormal states (those in which something has gone wrong); faulty states are also drawn with dashed lines, while the others represent normal configurations. Notice that the unique faulty state in this model is that named t. In this simple model, for every non-faulty execution, $p \wedge q$ is always true. In dCTL- this is expressed by the formula $\mathbf{O}(p \wedge q)$. Note that there also exist normal executions for which $p \wedge q \wedge r$ holds. This fact is expressed as $\mathbf{P}(p \wedge q \wedge r)$. Other deontic operators such as *prohibition* can be expressed by using those introduced above (see [8]).

One of the interesting characteristics of dCTL- is the possibility of distinguishing between formulas that state properties of good executions and the standard formulas, which state properties over all possible executions. For every formula φ, a formula φ^N can be built, which captures the same property as φ but restricted to good executions. This leads to the notion of *normative formula* of a given formula, and is defined as follows.

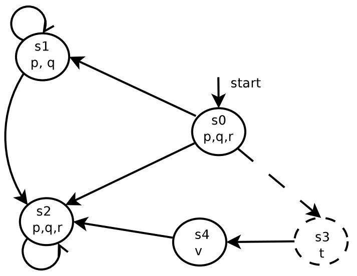


Fig. 1. A Simple Colored Kripke Structure

Definition 1. *Given a* dCTL*- formula* φ *over an alphabet AP, its* normative formula φ^N, *is defined by the following rules:*

- $(p_i)^N \stackrel{\mathsf{def}}{=} p_i$, $(\neg\varphi)^N \stackrel{\mathsf{def}}{=} \neg\varphi^N$, $(\varphi \wedge \varphi')^N \stackrel{\mathsf{def}}{=} \varphi^N \wedge \varphi'^N$,
- $(\mathsf{A}(\varphi\,\mathcal{U}\,\varphi'))^N \stackrel{\mathsf{def}}{=} \mathbf{O}(\varphi^N\,\mathcal{U}\,\varphi'^N)$, $(\mathsf{A}(\varphi\,\mathcal{W}\,\varphi'))^N \stackrel{\mathsf{def}}{=} \mathbf{O}(\varphi^N\,\mathcal{W}\,\varphi'^N)$,
- $(\mathsf{E}(\varphi\,\mathcal{U}\,\varphi'))^N \stackrel{\mathsf{def}}{=} \mathbf{P}(\varphi^N\,\mathcal{U}\,\varphi'^N)$, $(\mathsf{E}(\varphi\,\mathcal{W}\,\varphi'))^N \stackrel{\mathsf{def}}{=} \mathbf{P}(\varphi^N\,\mathcal{W}\,\varphi'^N)$,
- $(\mathbf{O}(\varphi\,\mathcal{U}\,\varphi'))^N \stackrel{\mathsf{def}}{=} \mathbf{O}(\varphi^N\,\mathcal{U}\,\varphi'^N)$, $(\mathbf{O}(\varphi\,\mathcal{W}\,\varphi'))^N \stackrel{\mathsf{def}}{=} \mathbf{O}(\varphi^N\,\mathcal{W}\,\varphi'^N)$,
- $(\mathbf{P}(\varphi\,\mathcal{U}\,\varphi'))^N \stackrel{\mathsf{def}}{=} \mathbf{P}(\varphi^N\,\mathcal{U}\,\varphi'^N)$, $(\mathbf{P}(\varphi\,\mathcal{W}\,\varphi'))^N \stackrel{\mathsf{def}}{=} \mathbf{P}(\varphi^N\,\mathcal{W}\,\varphi'^N)$.

3 (Bi)Simulations and Fault-Tolerance

In this section we present a number of simulation relations that allow us to capture various kinds of fault-tolerance, namely *masking*, *nonmasking*, and *fail-safe*. In order to define these relations, we follow the basic definitions regarding simulation and bisimulation relations given in [7]. Due to space restrictions, the technical proofs of the theorems presented in this section are omitted; the interested reader can find them in [10].

We will assume that the properties of interest of a system will be safety and liveness properties (recall that any temporal specification can be written as a conjunction of safety and liveness properties [1]). Basically, in order to check fault-tolerance, we consider two colored Kripke structures of a system, the first one acting as a specification of the intended behavior and the second as the fault-tolerant implementation. A system will be fault-tolerant if it is able to preserve, to some degree, the safety and liveness properties corresponding to its specification, even in the presence of faults. Our purpose will be to capture, via appropriate (bi)simulation relations between the system specification and the fault-tolerant implementation, different kinds of fault-tolerance, with different levels of property preservation.

In the following definitions, given a colored Kripke structure with a labeling L, we consider the notion of a sublabeling: we say that L_0 is a sublabeling of L (denoted by $L_0 \subseteq L$), if $L_0(s) = L(s) \cap AP'$, for all states s and some $AP' \subseteq AP$. We also say that L_0 is obtained by restricting AP to AP'. The concept of sublabeling allows us to focus on certain properties of models.

Let us start by introducing the notion of masking tolerance relations.

Definition 2. *(Masking fault-tolerance) Given two colored Kripke structures* $M = \langle S, I, R, L, \mathcal{N} \rangle$ *and* $M' = \langle S', I', R', L', \mathcal{N}' \rangle$*, we say that a relationship* $\prec_{Mask} \subseteq S \times S'$ *is* masking fault-tolerant *for sublabelings* $L_0 \subseteq L$ *and* $L'_0 \subseteq L'$ *iff:*

(A) $\forall s_1 \in I : (\exists s_2 \in I' : s_1 \prec_{Mask} s_2)$ *and* $\forall s_2 \in I' : (\exists s_1 \in I : s_1 \prec_{Mask} s_2)$.
(B) for all $s_1 \prec_{Mask} s_2$ *the following holds:*
  (1) $L_0(s_1) = L'_0(s_2)$.
  (2) if $s'_1 \in Post_N(s_1)$*, then there exists* $s'_2 \in Post_N(s_2)$ *with* $s'_1 \prec_{Mask} s'_2$.
  (3) if $s'_2 \in Post_N(s_2)$*, then there exists* $s'_1 \in Post_N(s_1)$ *with* $s'_1 \prec_{Mask} s'_2$.
  (4) if $s'_2 \in Post_F(s_2)$*, then either there exists* $s'_1 \in Post_N(s_1)$ *with* $s'_1 \prec_{Mask} s'_2$ *or* $s_1 \prec_{Mask} s'_2$.

We say that state s_2 is masking fault-tolerant for s_1 when $s_1 \prec_{Mask} s_2$. Intuitively, the intention in the definition is that, starting in s_2, faults can be masked in such a way that the behavior exhibited is the same as that observed when starting from s_1 and executing transitions without faults. Let us explain the above definition. Conditions A, $B.1$, $B.2$ and $B.3$ imply that we have a bisimulation between the normative parts of M and M'. Condition $B.4$ states that every outgoing faulty transition from s_2 either must be matched to an outgoing normal transition from s_1, or s'_2 is masking fault-tolerant for s_1.

Notice that, if there exists a self-loop at state s'_2, then we can stay forever satisfying $s_1 \prec_{Mask} s'_2$. Therefore, a fairness condition needs to be imposed on $B.4$ to ensure that masking implementations preserve liveness properties. For example, we can assume that, if a set of non-faulty transitions are enabled infinitely often, then these transitions will be executed infinitely often. We denote by $M \vDash_f \varphi$ the restriction of $\vDash$ to fair executions. It is worth remarking that the condition symmetric to ($B.4$) is not required, since we are only interested in the masking properties of M'.

We say that M' masks faults for M iff for every initial state s_0 of M there exists an initial state s'_0 of M' such that $s_0 \prec_{Mask} s'_0$, for some masking fault-tolerant relation $\prec_{Mask}$; we denote this situation by $M \prec_{Mask} M'$. Let us not present a simple example to illustrate the above definition.

Example 1. Let us consider a memory cell that stores a bit of information and supports reading and writing operations. A state in this system maintains the current value of the memory cell ($m = i$, for $i = 0, 1$), writing allows one to change this value, and reading returns the stored value.

A potential fault in this problem would be that the cell unexpectedly loses its charge, and its stored value turns into another one (e.g., it changes from 1 to 0 due to charge loss). A typical technique to deal with this situation is *redundancy*: use three memory bits instead of one. Writing operations are performed simultaneously on the three bits. Reading, on the other hand, returns the value that is repeated at least twice in the memory bits, known as *voting*, and the ready value is written back in all the bits.

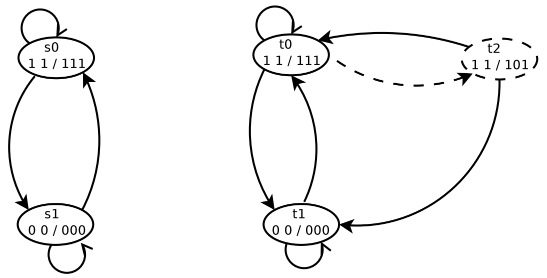


Fig. 2. Two masking fault-tolerance colored Kripke structures

In a model of this system, each state is described by variables m and w, which record the value stored in the system (taking *voting* into account) and the last writing operation performed, respectively. The state also maintains the values of the three bits that constitute the system, captured by boolean variables c_0, c_1 and c_2. For instance, in Figure 2, state s_0 contains the information 11/111, representing the state: $w = 1$, $m = 1$, $c_0 = 1$, $c_1 = 1$, and $c_2 = 1$.

Consider the colored Kripke structures M (left) and M' (right) depicted in Figure 2. M contains only normal transitions describing the expected ideal behavior (without taking into account faults). M' includes a model of a fault: a bit may suffer a discharge and then it changes its value from 1 to 0. It is straightforward to show that in this simple case there exists a masking fault-tolerance relation (specifically, the relation $R_1 = \{(s_0, t_0), (s_1, t_1), (s_0, t_2)\}$) between M and M' with the sublabelings L_0 and L'_0 obtained by restricting L and L' to propositions m and w, respectively.

An important property of masking fault-tolerance is that both safety and liveness properties of normative (i.e., non-faulty) executions are preserved by masking tolerant implementations under fairness restrictions.

Theorem 1. *Let $M = \langle S, I, R, L, \mathcal{N} \rangle$ and $M' = \langle S', I', R', L', \mathcal{N}' \rangle$ be colored Kripke structures, $s_1 \in S$ and $s_2 \in S'$. If $s_1 \prec_{Mask} s_2$ for sublabelings L_0 and L'_0 obtained by restricting L and L' to AP', respectively, then $M, s_1 \vDash_f \varphi^N \Rightarrow M', s_2 \vDash_f \varphi$, where all the propositional variables of φ are in AP'.*

Let us now focus on nonmasking fault-tolerance. This kind of tolerance is less strict than masking tolerance, since it allows for the existence of some states which do not mask faults. Intuitively, this type of fault-tolerance allows the system to violate its specification while it is recovering from a fault and returning to a normal behavior. More technically, the normative liveness properties of the system are always preserved, whereas the normative safety properties may not be fully preserved, but must be *eventually* restated. The characterization of this kind of fault-tolerance is the following.

Definition 3. *(Nonmasking fault-tolerance) Given two colored Kripke structures $M = \langle S, I, R, L, \mathcal{N} \rangle$ and $M' = \langle S', I', R', L', \mathcal{N}' \rangle$, we say that a relation $\prec_{Nonmask} \subseteq S \times S'$ is nonmasking for sublabelings $L_0 \subseteq L$ and $L'_0 \subseteq L'$, iff:*

(A) $\forall s_1 \in I : (\exists s_2 \in I' : s_1 \prec_{Nonmask} s_2)$ *and* $\forall s_2 \in I' : (\exists s_1 \in I : s_1 \prec_{Nonmask} s_2)$.

(B) for all $s_1 \prec_{Nonmask} s_2$ *the following holds:*

  (1) $L_0(s_1) = L'_0(s_2)$.

  (2) if $s'_1 \in Post_N(s_1)$*, then there exists* $s'_2 \in Post_N(s_2)$ *with* $s'_1 \prec_{Nonmask} s'_2$.

  (3) if $s'_2 \in Post_N(s_2)$*, then there exists* $s'_1 \in Post_N(s_1)$ *with* $s'_1 \prec_{Nonmask} s'_2$.

  (4) if $s'_2 \in Post_F(s_2)$*, then there exists* $s'_1 \in Post_N(s_1)$ *with* $s'_1 \prec_{Nonmask} s'_2$*, or*

  (5) if $s'_2 \in Post_F(s_2)$ *with* $s'_1 \not\prec_{Nonmask} s'_2$ *for all* $s'_1 \in Post_N(s_1)$*, then there exists a finite path fragment* $s_2 \dashrightarrow s'_2 \Rightarrow^* s''_2$ *such that either* $s'_1 \prec_{Nonmask} s''_2$ *for some* $s'_1 \in Post_N(s_1)$*, or* $s_1 \prec_{Nonmask} s'_2$.

Let us explain this definition. Conditions A, $B.1$, $B.2$, $B.3$, $B.4$ are similar to the conditions of Def. 2. Condition $B.5$ asserts that if $s_1 \prec_{Nonmask} s_2$ and every "faulty" successor state s'_2 of s_2 is not in a nonmasking relation with any normal successor of s_1, then there exists a path fragment that leads from s_2 to s''_2 such that $s'_1 \prec_{Nonmask} s''_2$ for some normal successor state s'_1 of s_1, or s'_2 is nonmasking fault-tolerant for s_1.

We say that M' is nonmasking fault-tolerant wrt. M iff for every initial state s_0 of M there exists an initial state s'_0 of M' such that $s_0 \prec_{Nonmask} s'_0$, for some nonmasking fault-tolerance $\prec_{Nonmask}$(indicated by $M \prec_{Nonmask} M'$).

At first sight, nonmasking fault-tolerance seems similar to the notion of weak bisimulation used in process algebra [2], where silent steps are taken into account. Notice however that, as opposed to weak bisimulation where silent steps produce only nonobservable (i.e., internal) changes, faults may produce observable changes in a nonmasking fault-tolerance relation. Let us present an example of nonmasking tolerance.

Example 2. For the memory cell introduced in Example 1, consider now the colored Kripke structures M (left) and M' (right) depicted in Figure 3. Now we consider that two faults may occur: up to two bits may lose its charge before any normal transition is taken. The relation $R_2 = \{(s_0, t_0), (s_1, t_1), (s_0, t_2)\}$ is nonmasking tolerant for (M, M') and the sublabelings L_0 and L'_0, obtained by restricting L and L' to propositions m and w, respectively.

An important property is that if s_2 is nonmasking fault-tolerant for s_1 and for every state of normal paths starting in s_1, φ holds, then in fair executions starting in s_2, φ eventually holds even in the presence of faults.

Theorem 2. *Let* $M = \langle S, I, R, L, \mathcal{N} \rangle$ *and* $M' = \langle S', I', R', L', \mathcal{N}' \rangle$ *be colored Kripke structures,* $s_1 \in S$ *and* $s_2 \in S'$*. If* $s_1 \prec_{Nonmask} s_2$ *for sublabelings* L_0 *and* L'_0 *obtained by restricting* L *and* L' *to* AP'*, respectively, then* $M, s_1 \models_f \varphi^N \Rightarrow M', s_2 \models_f \mathsf{AFAG}(\varphi)$*, where all the propositional variables of* φ *are in* AP'.

We now present a characterization of failsafe fault-tolerance. Essentially, failsafe fault-tolerance must ensure that the system will stay in a safe state, although it may be limited in its capacity. More technically, this means that the normative safety properties must be preserved, while normative liveness properties may not be respected.

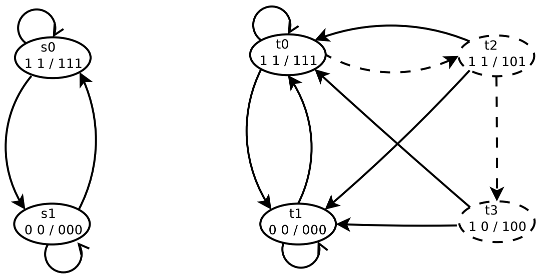


Fig. 3. Two nonmasking fault-tolerance colored Kripke structures

Definition 4. *(Failsafe fault-tolerance) Given two colored Kripke structures* $M = \langle S, I, R, L, \mathcal{N} \rangle$ *and* $M' = \langle S', I', R', L', \mathcal{N}' \rangle$*, we say that a relation* $\prec_{Failsafe} \subseteq S \times S'$ *is failsafe for sublabelings* $L_0 \subseteq L$ *and* $L'_0 \subseteq L'$ *iff:*

(A) $\forall s_1 \in I : (\exists s_2 \in I' : s_1 \prec_{Failsafe} s_2)$ *and* $\forall s_2 \in I' : (\exists s_1 \in I : s_1 \prec_{Failsafe} s_2)$.

(B) for all $s_1 \prec_{Failsafe} s_2$ *the following holds:*

- *(1)* $L_0(s_1) = L'_0(s_2)$.
- *(2) if* $s'_1 \in Post_N(s_1)$*, then there exists* $s'_2 \in Post_N(s_2)$ *with* $s'_1 \prec_{Failsafe} s'_2$.
- *(3) if* $s'_2 \in Post_N(s_2)$*, then there exists* $s'_1 \in Post_N(s_1)$ *with* $s'_1 \prec_{Failsafe} s'_2$.
- *(4) if* $s'_2 \in Post_F(s_2)$*, then either there exists* $s'_1 \in Post_N(s_1)$ *with* $L_0(s'_1) = L'_0(s'_2)$ *or* $L_0(s_1) = L'_0(s'_2)$.

Whenever two states s_1 and s_2 are related by a failsafe fault-tolerant relation $\prec_{Failsafe}$, i.e., $s_1 \prec_{Failsafe} s_2$, we say that s_2 is failsafe fault-tolerant for s_1. We say that M' is failsafe fault-tolerant for M iff for every initial state s_0 of M there exists an initial state s'_0 of M' such that $s_0 \prec_{Failsafe} s'_0$, for some failsafe fault-tolerant relation $\prec_{Failsafe}$; we denote this situation by $M \prec_{Failsafe} M'$.

Let us explain the above definition. Conditions A, $B.1$, $B.2$, and $B.3$ are similar to those of Def. 2, regarding masking fault-tolerance. Condition $B.4$ states that if $s_1 \prec_{Failsafe} s_2$, then every outgoing faulty transition from s_2 either must be matched to an outgoing normal transition from s_1, requiring states s'_1 and s'_2 to be labeled with the same propositions, or s'_2 must be failsafe fault-tolerant for s_1. We now present a simple example to illustrate this notion.

Example 3. Consider the colored Kripke structures M (left) and M' (right) depicted in Figure 4. M is the specification of the expected ideal, fault-free, behavior. M', on the other hand, involves the occurrence of one fault. The relation $R_3 = \{(s_0, t_0), (s_1, t_1)\}$ is a failsafe fault-tolerance relation for (M, M') and the sublabelings that are obtained by restricting L and L' to propositions m and w.

Our definition of failsafe fault-tolerance preserves safety properties.

Theorem 3. *Let* $M = \langle S, I, R, L, \mathcal{N} \rangle$ *and* $M' = \langle S', I', R', L', \mathcal{N}' \rangle$ *be colored Kripke structures,* $s_1 \in S$ *and* $s_2 \in S'$. *If* $s_1 \prec_{Failsafe} s_2$ *for sublabelings* L_0

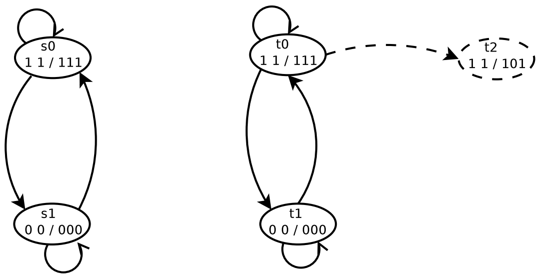


Fig. 4. Two failsafe fault-tolerance colored Kripke structures

and L'_0 obtained by restricting L and L' to AP', respectively, and φ is a safety property, then $M, s_1 \vDash \varphi^N \Rightarrow M', s_2 \vDash \varphi$, where all the propositional variables of φ are in AP'.

This property says that if we have a failsafe relation between s_1 and s_2 and for every state in normal paths starting in s_1, φ holds in the absence of faults, then φ is always true even in the presence of faults in paths starting in s_2.

The following lemma provides important information regarding all the fault-tolerance relations defined above.

Lemma 1. *Given relations $\prec_{Mask}$, $\prec_{Nonmask}$ and $\prec_{Failsafe}$, we have the following properties:*

- *$\prec_{Mask}$, $\prec_{Nonmask}$, $\prec_{Failsafe}$ are transitive,*
- *If M does not have faults, then: $M \sqsubset M' \Rightarrow M' \sqsubset M$, where $\sqsubset \in \{\prec_{Mask}, \prec_{Nonmask}, \prec_{Failsafe}\}$*
- *$\prec_{Mask}$, $\prec_{Nonmask}$ and $\prec_{Failsafe}$ are not necessarily reflexive.*

We also have properties of these relations corresponding to *inclusions*:

Theorem 4. *Let Mask, NMask and FSafe be the sets of masking, nonmasking and failsafe relations between two colored Kripke structures M and M'. Then we have:*

$$\mathsf{Mask} \subseteq \mathsf{FSafe} \textit{ and } \mathsf{Mask} \subseteq \mathsf{NMask}$$

3.1 Checking Fault-Tolerance Properties

Simulation and bisimulation relations are amenable to efficient computational treatment. For instance, in [7, 11] algorithms for calculating several simulation and bisimulation relations are described and proved to be polynomial with respect to the number of states and transitions of the corresponding models. We have adapted these algorithms to our setting, thus obtaining efficient procedures to prove masking, nonmasking and failsafe fault-tolerance. Such algorithms can be used to verify whether $M \sqsubset M'$, with $\sqsubset \in \{\prec_{Mask}, \prec_{Nonmask}, \prec_{Failsafe}\}$. We discuss the algorithm for computing masking fault-tolerance. The algorithms

Algorithm 1. Computation of masking fault-tolerant

```
Input: colored Kripke structure $M$
Output: masking fault-tolerant $\prec_{Mask}$
 1: for all $s_2 \in \mathcal{F}$ do
 2:    $Mask(s_2) := \{s_1 \in \mathcal{N} \mid L_0(s_1) = L_0(s_2)\}$
 3:    $Remove(s_2) := \mathcal{N} \backslash Pre_N(Mask(s_2))$
 4: end for
 5: while $\exists\ s_2' \in \mathcal{F}$ with $Remove(s_2') \neq \emptyset$ do
 6:    select $s_2'$ such that $Remove(s_2') \neq \emptyset$
 7:    for all $s_1 \in Remove(s_2')$ do
 8:       for all $s_2 \in Pre_N(s_2')$ do
 9:          if $s_1 \in Mask(s_2)$ then
10:             $Mask(s_2) := Mask(s_2) \backslash \{s_1\}$
11:             for all $s \in Pre_N(s_1)$ with $Post_N(s) \cap Mask(s_2) = \emptyset$ do
12:                $Remove(s_2) := Remove(s_2) \cup \{s\}$
13:             end for
14:          end if
15:       end for
16:    end for
17:    $Remove(s_2') := \emptyset$
18: end while
19: for all $s_2 \in \mathcal{F}$ do
20:    if $Post^*(s_2) \neq Post^*(Mask(s_2))$ then
21:       $Mask(s_2) := \emptyset$
22:    end if
23: end for
24: return $\{(s_1, s_2) \mid s_1 \in Mask(s_2)\}$
```

for the other relations can be obtained in a similar way; the interested reader is referred to [10]. The basic scheme for checking masking fault-tolerance is sketched in Algorithm 1. This algorithm takes as input a colored Kripke structure $M = \langle S, I, R, L, \mathcal{N} \rangle$ and a sublabeling $L_0 \subseteq L$, and produces a masking fault-tolerance relation $\prec_{Mask}$. In order to check fault-tolerance properties, we take two colored Kripke structures M and M' over AP (the system specification and the fault-tolerant implementation), and combine them in a single structure $M \oplus M'$ via disjoint union, to feed as input to the algorithm. Notice that Algorithm 1 only deals with the case of faulty states/transitions (condition $B.4$ of Def. 2), since a standard bisimulation algorithm can be used for checking that the normative behavior described in the specification is bisimilar with that exhibited by the fault-tolerant implementation (conditions $B.2$ and $B.3$). This algorithm is an adaptation of the similarity-checking algorithm for finite graphs defined in [11]. We have made slight changes (between lines 1 and 18) in order to explore all the faulty transitions $s_2 \dashrightarrow s_2'$ and look for those normal transitions $s_1 \rightarrow s_1'$ which mask faulty ones.

Let us briefly explain Algorithm 1. Consider $\mathcal{F} = S \backslash \mathcal{N}$ to be the set of faulty states in M. For each $s_2 \in \mathcal{F}$, the set $Mask(s_2)$ contains the normal states that are candidates for masking s_2. Initially, $Mask(s_2)$ consists of all normal states with the same labels as s_2 and $Remove(s_2)$ contains all the normal states which do not have a (normal) successor state masking s_2. Moreover, these states cannot mask any of the predecessors of s_2. The termination condition of the outermost loop of lines 5–18 is $Remove(s_2') = \emptyset$ for all $s_2' \in \mathcal{F}$, in which case there are no normal states that need to be removed from the sets of simulators $Mask(s_2)$ for $s_2 \in Pre_N(s_2')$. Within the while-loop body, the main idea is to pick one pair (s_1, s_2') with $s_1 \in Remove(s_2')$ per iteration; for each one we scan through the predecessor list of s_2' and test for each normal state $s_2 \in Pre_N(s_2')$ to see whether $s_1 \in Mask(s_2)$. In the positive case, s_1 is removed from $Mask(s_2)$. Subsequently, we add to the set $Remove(s_2)$ all normal predecessors s of s_1 such that $Post_N(s) \ \cap \ Mask(s_2) = \emptyset$. The last for-loop checks whether from any faulty state s_2 which is masked by a normal state, we can reach a normal state to recover to the normal behavior. Therefore, for each faulty state s_2, we inspect the existence of the reachable normal states from s_2 (line 20). In the case that the set of successors of s_2 is not equal to the set of the normal successors of the normal state which mask s_2, then we remove all states from $Mask(s_2)$. Finally, the masking fault-tolerance $\prec_{Mask}$ is obtained from a colored Kripke structure M by performing the union between the set obtained from a bisimulation algorithm used to check that the system strongly bisimulates the specification for the normative part, and the set returned by Algorithm 1.

Regarding the complexity of the algorithm for checking masking fault-tolerance, it is polynomial. More precisely, the time complexity of the bisimulation quotient algorithm [7] is $\mathcal{O}(|\mathcal{N}| \cdot |AP'| + E \cdot \log\ |\mathcal{N}|)$, where E is the number of edges in M. On the other hand, Algorithm 1 can be computed in time $\mathcal{O}(E \cdot |\mathcal{F}| + |\mathcal{F}| \cdot AP' + |\mathcal{F}|)$. Hence, the masking fault-tolerance of a colored Kripke structure $M = \langle S, I, R, L, \mathcal{N} \rangle$ for sublabeling $L_0 \subseteq L$ obtained by restricting AP to AP', can be computed in a running time of $\mathcal{O}(E \cdot \log\ |\mathcal{N}| + E \cdot |F|)$.

4 An Example: The Muller C-element

The Muller C-element [14] is a simple delay-insensitive circuit which contains two boolean inputs and one boolean output. Its logical behavior is described as follows: if both inputs are true (resp. false) then the output of the C-element becomes true (resp. false). If the inputs do not change, the output remains the same. In [3], the following (informal) specification of the C-element with inputs x and y and output z is given:

> *(i) Input x (resp. y) changes only if $x \equiv z$ (resp., $y \equiv z$), (ii) Output z becomes true only if $x \wedge y$ holds, and becomes false only if $\neg x \wedge \neg y$ holds; (iii) Starting from a state where $x \wedge y$, eventually a state is reached where z is set to the same value that both x and y have. Ideally, both x and y change simultaneously. Faults may delay changing either x or y.*

We consider an implementation of the C-element with a majority circuit involving three inputs, where an extra input u in the circuit is added. Then, the

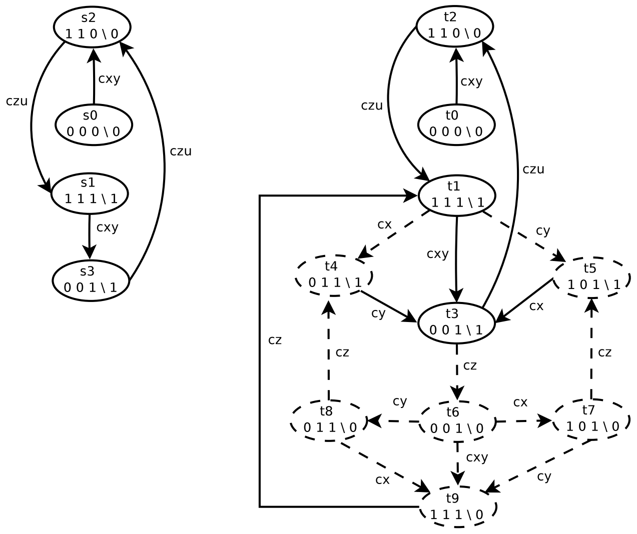


Fig. 5. A nonmasking fault-tolerance for the Muller C-element with a majority circuit

predicate $maj(x, y, u)$ returns the value of the majority circuit, which is assumed to work correctly, and is defined as $maj(x, y, u) = (x \wedge y) \vee (x \wedge u) \vee (y \wedge u)$. In addition to the traditional logical behavior of the C-element, u and z have to change at the same time, where the output z is fed back to the input u. Figure 5 shows two models of this circuit. M exhibits the ideal behavior of the C-element containing only normal transitions. M' takes into account the possibility of faults occurring, and provides a reaction to these. Every state in these models is composed of boolean variables x, y, u, and z, where x, y, and u represent the inputs, and z represents the output. For instance, the state s_0 contains the information $000 \setminus 0$ interpreted (reading from left to right) as $x = 0$, $y = 0$, $u = 0$, and $z = 0$. Transitions are labeled by subsets of the set $\{cx, cy, cu, cz\}$ of actions; action cx (resp., cy and cu) is the action that changes input x (resp., y and u); cz is the action of changing output z. When the actions cx and cy are executed in the same transition, we just write cxy. We consider two types of faults: *(i)* a delay may occur in the arrival of some of the inputs x or y (i.e., they do not change simultaneously), and *(ii)* a delay in the signal from z to u occurs. We can observe these classes of faults in the faulty states (indicated by dashed circles) when either x and y or u and z do not match one another. The relation $R_{c-element} = \{(s_0, t_0), (s_1, t_1), (s_2, t_2), (s_3, t_3)\}$ is a nonmasking fault-tolerance for (M, M') and the sublabelings obtained by restricting the original labelings to letters u, x, y, z. Therefore, when the majority circuit behaves correctly, this implementation masks delays of inputs.

We have developed other case studies illustrating the practical application of our framework, based on well known fault-tolerance models, including, e.g., Byzantine agreement. These can be found in [10].

5 Related Work

Our work is most closely related to formal approaches to fault-tolerance. One of these is that presented in [4], where the problem of multitolerance is addressed. In order to do so, the authors define the concepts of masking, nonmasking and failsafe tolerances using liveness and safety specifications in a linear-time framework. In our approach, we focus on branching time properties of programs. In our opinion, branching time is important for fault-tolerance specification. This view is also shared by Attie, Arora, and Emerson in [6], where an algorithm for synthesizing fault-tolerant programs from CTL specifications is presented. They consider CTL as the temporal logic specification for the input of their synthesis method. Instead of using CTL, we use a branching time temporal logic that has a convenient mechanism for stating fault-tolerance properties, via the use of deontic operators. We believe that our formalism is better suited for capturing fault-tolerance properties. Finally, we can mention the works presented in [12, 13], where various notions of bisimulation are investigated with the aim of capturing fault-tolerant properties, in the context of process algebras. An obvious difference wrt. our work is that we use a state based approach and a temporal logic to reason about state based models, in contrast to the aforementioned works where process algebras are employed for modeling systems, and the associated logic is a variation of Hennesy-Milner logic, which is known to be less expressive than temporal logics. Also, the notions of masking, nonmasking and failsafe fault-tolerance are not investigated in the referenced works.

6 Conclusions and Future Work

We have presented a characterization of different levels of fault-tolerance by means of simulation relations. This formalization is simple and uses standard notions of simulation relations, by relating an operational system specification and a corresponding fault-tolerant implementation. Moreover, our approach to capturing fault-tolerance enables us to automatically verify, for example, that a given implementation of a system masks certain faults, or recovers from these faults, by employing variants of traditional bisimulation algorithms to our context. Indeed, we have adapted well known (bi)simulation algorithms to our setting, so that one can automatically check if a system implementation exhibits some degree of fault-tolerance. We have also studied the complexity of the resulting algorithms, and proved that they preserve the time complexity of traditional bisimulation algorithms. We have also studied properties of our formalizations of fault-tolerance, showing that different kinds of temporal properties are preserved, depending on the degree of fault-tolerance that a system exhibits. Moreover, we have also presented results relating the different kinds of fault-tolerance.

As future work, we are exploring the extension of our setting with *synthesis*, so that fault-tolerant programs may be automatically constructed from the system specification, a description of the faults and their consequences, and a desired degree of fault-tolerance. Synthesis of programs has been extensively investigated in the context of linear time logic [15], while in our case it is necessary to deal with a branching time formalism. This is important since, as argued in [6], some important properties related to fault-tolerance require branching time operators. It is also in [6] where a framework for synthesis of programs from branching time specifications is introduced. We believe that some of the work presented in this paper can be used to extend that introduced in [6], to automatically synthesize programs that mask faults, recover from error situations or stay in safe states. Finally, we also plan to extend our framework to accommodate multitolerance [4], in which multiple classes of faults may occur simultaneously.

Acknowledgements. The authors would like to thank the anonymous referees for their helpful comments. This work was partially supported by a Fellowship from IBM Canada, in support of the Automotive Partnership Canada funded project NECSIS; by the Argentinian Agency for Scientific and Technological Promotion (ANPCyT), through grants PICT PAE 2007 No. 2772, PICT 2010 No. 1690 and PICT 2010 No. 2611; and by the MEALS project (EU FP7 programme, grant agreement No. 295261).

References

1. Alpern, B., Schneider, F.: Defining Liveness. Inf. Process. Lett. 21(4) (1985)
2. Milner, R.: Communication and Concurrency. PHI Series in Computer Science. Prentice-Hall (1989)
3. Arora, A., Gouda, M.: Closure and Convergence: A Foundation of Fault-Tolerant Computing. IEEE Trans. Soft. Eng. 19(11) (1993)
4. Arora, A., Kulkarni, S.: Component Based Design of Multitolerant Systems. IEEE Trans. Software Eng. 24(1) (1998)
5. Arora, A., Kulkarni, S.: Detectors and Correctors: A Theory of Fault-Tolerance Components. In: Proc. of ICDCS (1998)
6. Attie, P., Arora, A., Emerson, A.: Synthesis of fault-tolerant concurrent programs. ACM Trans. Program. Lang. Syst. 26(1) (2004)
7. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press (2008)
8. Castro, P.F., Kilmurray, C., Acosta, A., Aguirre, N.: dCTL: A Branching Time Temporal Logic for Fault-Tolerant System Verification. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 106–121. Springer, Heidelberg (2011)
9. Cristian, F.: A rigorous approach to fault-tolerant programming. IEEE Trans. Software Eng. (1985)
10. Demasi, R., Castro, P., Maibaum, T., Aguirre, N.: Characterizing Fault-Tolerant Systems by Means of Simulation Relations, Tech. Report, `http://www.cas.mcmaster.ca/~demasira/reportSimFTS.pdf`
11. Henzinger, M., Henzinger, T., Kopke, P.: Computing Simulations on Finite and Infinite Graphs. In: Proc. of FOCS (1995)
12. Janowski, T.: Bisimulation and Fault-Tolerance. PhD thesis (1995)
13. Janowski, T.: On Bisimulation, Fault-Monotonicity and Provable Fault-Tolerance. In: Proc. of *AMAST* (1997)
14. Mead, C., Conway, L.: Introduction to VLSI systems. Addison-Wesley (1980)
15. Pnueli, A., Rosner, R.: On the Synthesis of a Reactive Module. In: Proc. of POPL (1989)

Author Index

Abrial, Jean-Raymond 1
Aguirre, Nazareno 428
Albert, Elvira 285
André, Étienne 331
Andriamiarina, Manamiary Bruno 268

Bogdanov, Kirill 253

Cabrera Castillos, Kalou 316
Castro, Pablo F. 428
Choppy, Christine 331
Ciobâcă, Ştefan 347
Correas, Jesús 285
Curzon, Paul 238

Dadeau, Frédéric 316
Demasi, Ramiro 428
Derrick, John 253
Dima, Catalin 61
Dong, Jin Song 162, 331

Fortin, Jean 124
Frappier, Marc 223

Gava, Frédéric 124
Gesell, Manuel 177
Giachino, Elena 394
Graf, Susanne 77
Grazia, Carlo A. 394
Guedj, Michael 124
Gui, Lin 162
Guldstrand Larsen, Kim 154

Harrison, Michael D. 238
Heather, James 94
Hoang, Thai Son 16
Hudon, Simon 16

Ishii, Daisuke 139

Julliand, Jacques 316

Kanso, Bilal 316

Lanet, Jean-Louis 223
Laneve, Cosimo 301, 394
Lang, Frédéric 61
Lausdahl, Kenneth 46
Lienhardt, Michael 394
Liu, Shuang 331
Liu, Yang 162, 331
Lucanu, Dorel 362

Maibaum, Thomas S.E. 428
Mateescu, Radu 61
Melquiond, Guillaume 139
Méry, Dominique 208, 268
Millo, Jean-Vivien 109
Moran, Murat 94
Morgenstern, Andreas 177

Nakajima, Shin 139
Narwane, Ganesh Khandu 109
Nielson, Flemming 412
Nielson, Hanne Riis 412

Olderog, Ernst-Rüdiger 378

Padovani, Luca 301
Poppleton, Michael 208
Prehofer, Christian 31
Puebla, Germán 285

Quinton, Sophie 77

Ramesh, S. 109
Román-Díez, Guillermo 285
Rukšėnas, Rimvydas 238
Rusu, Vlad 362

Savary, Aymerick 223
Schneider, Klaus 177
Schneider, Steve 94
Shankara Narayanan, Krishna 109
Singh, Neeraj Kumar 268
Song, Fu 192
Song, Songzheng 162
Sun, Jun 162, 331
Swaminathan, Mani 378

Taha, Safouan 316
Taylor, Ramsay 253
Touili, Tayssir 192

Vekris, Dimitris 61
Vigo, Roberto 412

Wadhwa, Bimlesh 331
Wong, Peter Y.H. 394