

Jim Davies
Leila Silva
Adenilso Simao (Eds.)

LNCS 6527

Formal Methods: Foundations and Applications

13th Brazilian Symposium on Formal Methods, SBMF 2010
Natal, Brazil, November 2010
Revised Selected Papers



Springer

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Alfred Kobsa

University of California, Irvine, CA, USA

Friedemann Mattern

ETH Zurich, Switzerland

John C. Mitchell

Stanford University, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz

University of Bern, Switzerland

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

TU Dortmund University, Germany

Madhu Sudan

Microsoft Research, Cambridge, MA, USA

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max Planck Institute for Informatics, Saarbruecken, Germany

Jim Davies Leila Silva Adenilso Simao (Eds.)

Formal Methods: Foundations and Applications

13th Brazilian Symposium on Formal Methods, SBMF 2010
Natal, Brazil, November 8-11, 2010
Revised Selected Papers

Volume Editors

Jim Davies
Oxford University, Department of Computer Science
Oxford OX1 3QD, UK
E-mail: Jim.Davies@comlab.ox.ac.uk

Leila Silva
Universidade Federal de Sergipe
Departamento de Ciência da Computação e Estatística
CEP 49100-000, Aracaju, SE, Brazil
E-mail: lmas@ufs.br

Adenilso Simao
Avenida Trabalhador São-Carlense, 400 Centro
13566-590, São Carlos, SP, Brazil
E-mail: adenilso@icmc.usp.br

ISSN 0302-9743
ISBN 978-3-642-19828-1
DOI 10.1007/978-3-642-19829-8
Springer Heidelberg Dordrecht London New York

e-ISSN 1611-3349
e-ISBN 978-3-642-19829-8

Library of Congress Control Number: 2011922662

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

LNCS Sublibrary: SL 2 – Programming and Software Engineering

© Springer-Verlag Berlin Heidelberg 2011

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

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

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

Printed on acid-free paper

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

Preface

This volume contains the papers presented at SBMF 2010: the 13th Brazilian Symposium on Formal Methods, organized during the week of November 8, 2010. The conference was held, for the second time, in the city of Natal, Rio Grande do Norte, Brazil, co-located with ICTSS 2010, the 22nd IFIP International Conference on Testing Software and Systems, and SAST 2010, the Brazilian Workshop on Systematic and Automated Software Testing.

The conference programme included three invited talks, given by Constance Heitmeyer (Naval Research Lab, USA), Bill Roscoe (University of Oxford, UK) and David Naumann (Stevens Institute of Technology, USA). It also included two workshops: *Using BOOGIE 2 in the Verification of Spec# Programs*, organized by K. Rustan M. Leino (Microsoft Research) and Rosemary Monahan (National University of Ireland), and *Workshop on B Dissemination (WOBD)*, chaired by Thierry Lecomte (ClearSy, France) on behalf of the DEPLOY project.

There was also an accompanying doctoral research symposium, with presentations from research students working on new developments in the theory and practice of formal methods, and a special session on the development of the formal methods curriculum.

Awards were made to: Wojciech Mostowski and Erik Poll, for the best paper, “Midlet Navigation Graphs in JML”; to Alexandra Silva, for the best paper presentation; to Giselle Reis, for the best doctoral presentation; to Tiago Massoni, for the best use of presentation technology; and to Rolf Hennicker, for the best contribution to the discussions that followed each author’s presentation.

A total of 18 research papers were presented at the conference, selected from 55 submissions, and included in revised form in this volume. We are grateful to the Programme Committee, and the additional reviewers, for their hard work in evaluating submissions and suggesting improvements. The papers were presented, by their authors, in seven separate sessions; these sessions were well attended, and we are grateful to the many participants who made additional, thoughtful contributions between, during, and after the paper presentations.

We are grateful to the organizers of this year’s conference, the Departamento de Informática e Matemática Aplicada of Rio Grande do Norte (UFRN) and the Brazilian Computer Society (SBC), and also to the sponsors: CNPq, the Brazilian Scientific and Technological Research Council; CAPES, the Brazilian Higher Education Funding Council; The Federal University of Rio Grande do Norte (UFRN); Miranda Computação e Comércio Ltda; SETIRN.

December 2010

Jim Davies
Leila Silva
Adenilso Simão

Organization

Programme Committee

Aline Andrade	Stephan Merz
David Aspinall	Alvaro Moreira
Luis Barbosa	Anamaria Moreira
Roberto Bigonha	Carroll Morgan
Michael Butler	Alexandre Mota
Andrew Butterfield	Arnaldo Moura
Ana Cavalcanti	David Naumann
Marcio Cornelio	Daltro Jose Nunes
Andrea Corradini	Jose Oliveira
Jim Davies (Co-chair)	Marcel Oliveira (Local Chair)
David Deharbe	Alberto Pardo
Ewen Denney	Alexandre Petrenko
Clare Dixon	Montréal, Canada
Rohit Gheyi	Leila Ribeiro
Rolf Hennicker	Augusto Sampaio
Juliano Iyoda	Leila Silva (Co-chair)
Zhiming Liu	Adenilso Simão (Co-chair)
Gerald Luetzgen	Heike Wehrheim
Patricia Machado	Jim Woodcock
Ana de Melo	

Additional Reviewers

Ludwig Adam	Charles Morisset
Renato Alexandre Silva	Regina Motz
Wilkerson L. Andrade	Stan Rosenberg
Tigran Avanesov	Asieh Salehi Fathabadi
Sebastian Bauer	Paulo Salem da Silva
Karine Birnfeld	Luis Sierra
Filippo Bonchi	Volker Stolz
Adilson Bonifácio	Ivan Tierno
Florent Bouchy	Jan Tobias Muehlberg
Alexander Ditter	Walter Vogler
Arnaud Dury	Shuling Wang
Adriano Gomes	James Welch
Bruno Gomes	Mar Yah Said
Rolf Hennicker	Sanaz Yeganefard
Giovanni Lucero	Jiaqi Zhu
Hugo Macedo	

Table of Contents

Directed Model Checking for B: An Evaluation and New Techniques	1
<i>Michael Leuschel and Jens Bendisposto</i>	
Midlet Navigation Graphs in JML	17
<i>Wojciech Mostowski and Erik Poll</i>	
Runtime Verification for Generic Classes with CONGU2	33
<i>Pedro Crispim, Antónia Lopes, and Vasco T. Vasconcelos</i>	
A High-Level Language for Modeling Algorithms and Their Properties	49
<i>Sabina Akhtar, Stephan Merz, and Martin Quinson</i>	
A Formal Environment Model for Multi-Agent Systems	64
<i>Paulo Salem da Silva and Ana C.V. de Melo</i>	
A Modal Interface Theory with Data Constraints	80
<i>Sebastian S. Bauer, Rolf Hennicker, and Michel Bidoit</i>	
Synchronizing Model and Program Refactoring	96
<i>Tiago Massoni, Rohit Gheyi, and Paulo Borba</i>	
A Type-Theoretic Framework for Certified Model Transformations	112
<i>Daniel Calegari, Carlos Luna, Nora Szasz, and Álvaro Tasistro</i>	
Simulating Truly Concurrent CSP	128
<i>Moritz Kleine and J.W. Sanders</i>	
Statistical Verification of Probabilistic Properties with Unbounded Until	144
<i>Håkan L.S. Younes, Edmund M. Clarke, and Paolo Zuliani</i>	
Reasoning about Assignments in Recursive Data Structures	161
<i>Alejandro Tamalet and Ken Madlener</i>	
Specification of a Localization Component Driven by a Goal-Based Approach: Some Lessons We Learned	177
<i>Abderrahman Matoussi, Frédéric Gervais, and Régine Laleau</i>	
A Formal Framework for Specifying and Analyzing Logs as Electronic Evidence	194
<i>Eduardo Mazza, Marie-Laure Potet, and Daniel Le Métayer</i>	

Formal Development of a Cardiac Pacemaker: From Specification to Code.....	210
<i>Artur O. Gomes and Marcel V.M. Oliveira</i>	
A Decision Procedure for Bisimilarity of Generalized Regular Expressions	226
<i>Marcello Bonsangue, Georgiana Caltais, Eugen-Ioan Goriac, Dorel Lucanu, Jan Rutten, and Alexandra Silva</i>	
Normalization of Linear Horn Clauses	242
<i>Thomas Martin Gawlitza, Helmut Seidl, and Kumar Neeraj Verma</i>	
A Graph-Based Implementation for Mechanized Refinement Calculus of OO Programs	258
<i>Zhiming Liu, Charles Morisset, and Shuling Wang</i>	
Automating Refinement of <i>Circus</i> Programs	274
<i>Frank Zeyda and Ana Cavalcanti</i>	
Author Index	291

Directed Model Checking for B: An Evaluation and New Techniques

Michael Leuschel and Jens Bendisposto

Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
{leuschel,bendisposto}@cs.uni-duesseldorf.de

Abstract. PROB is a model checker for high-level formalisms such as B, Event-B, CSP and Z. PROB uses a mixed depth-first/breadth-first search strategy, and in previous work we have argued that this can perform better in practice than pure depth-first or breadth-first search, as employed by low-level model checkers. In this paper we present a thorough empirical evaluation of this technique, which confirms our conjecture. The experiments were conducted on a wide variety of B and Event-B models, including several industrial case studies. Furthermore, we have extended PROB to be able to perform directed model checking, where each state is associated with a priority computed by a heuristic function. We evaluate various heuristic functions, on a series of problems, and find some interesting candidates for detecting deadlocks and finding specific target states.

Keywords: Model Checking, B-Method, Tool Support, Directed Model Checking, Search, Industrial Case Studies, SPIN.

1 Introduction

Many model checking tools, such as SMV [21,3] and SPIN [11,13,2], work on relatively low-level formalisms. Recently, however, there have also been model checkers which work on higher-level formalisms, such as TLC [25] for TLA⁺, FDR [9] for CSP and ALLOY [16] for a formalism of the same name (although the latter two are strictly speaking not model checkers). Another example is PROB [19,20] which accepts B [1].

It is relatively clear that a higher level specification formalism enables a more convenient modelling. On the other hand, conventional wisdom would dictate that a lower-level formalism will lead to more efficient model checking. However, our own experience has been different. During previous teaching and research activities, we have accumulated anecdotal evidence that using a high-level formalism such as B *can* be much more productive than using a low-level formalism such as Promela. The study [24,23] examined the elaboration of B models for ProB and Promela models for SPIN on ten different problems. Unsurprisingly, the time required to develop the Promela models was markedly higher than for the B models, (and some models could not be fully completed in Promela). The

study also found out that in practice both model checkers PROB and SPIN were comparable in model checking performance, despite PROB working on a much higher-level input language and being much slower when looking purely at the number of states that can be stored and processed per time unit. Other independent experimental evaluations also report good performance of PROB compared against SMV [15].

In [17] we first tried to analyse and understand this counter-intuitive fact. One explanation was that pure depth-first as employed by SPIN and other low-level model checkers fares very badly in the context of large state spaces. Similarly, a pure breadth-first strategy has problems in detecting long counter examples. We argued in [17] that PROB's mixed depth-first/breadth-first search enabled it to effectively find a larger class of errors. In this paper we test this conjecture empirically on a large number of B specifications. In addition, we present a new directed model checking algorithm for PROB: rather than randomly choosing between doing a depth-first or breadth-first step, we associate priorities with the pending states of the model checker. We then evaluate various ways of computing priorities on the same specifications.

In Section 2 we present the motivation for mixed depth-first/breadth-first search in more detail, and in Section 3 we perform a thorough empirical evaluation. In Section 4 we present the new directed model checking algorithm of PROB, along with a range of heuristic functions with their empirical evaluation. We finish with more related work and a conclusion in Section 5.

2 Combining Depth-First and Breadth-First for Improved Model Checking

In [17] we first tried to analyse and understand the counter-intuitive behaviour described above. Below, we recall some of the conclusions from [17]. One tricky issue is the much finer granularity of low-level models. If one is not careful, the number of reachable states can explode exponentially, compared to a corresponding high-level model. When writing Promela models, for example, great care has to be taken to make use of `atomic` (or even `dstep`) primitives and resetting dead temporary variables to default values. However, restrictions of `atomic` make it sometimes very difficult or impossible to hide all of the intermediate states. More details can be found in [17].

Searching for Errors in Large State Spaces. Let us disregard the granularity issue and let us look at simple problems, with simple datatypes, which can be easily translated from B to Promela, so that we have a one-to-one correspondence of the states of the models. In such a setting, one would assume that the SPIN model checker for Promela will outperform PROB by several orders of magnitude. Indeed, SPIN generates a specialised model checker in C which is then compiled, whereas PROB uses an interpreter written in Prolog. Furthermore, SPIN has accrued many optimisations over the years, such as partial order reduction [14,22]

and bitstate hashing [12]. However, even in this setting, this advantage of SPIN does not necessarily translate into better performance for real-life scenarios, in particular when using the model checker as a debugging tool for software systems, i.e., trying to find errors in a very large state space.

One experiment reported on in [17] is the NastyVendingMachine. It has a very large state space, where there is a systematic error in one of the operations of the model (as well as a deadlock when all tickets have been withdrawn). To detect the error, it is important to exercise this operation repeatedly. It is not important to generate long traces of the system, but it is important to systematically execute combinations of the individual operations. This explains why depth-first behaves so badly on this model, as it will always try to exercise the first operation of the model first. Note that a very large state space is a typical situation in software verification (sometimes the state space is even infinite).

Fortunately, SPIN provides a breadth-first option, with which it then finds the above error very quickly. However, for another class of problems, breadth-first fares badly. Indeed, in a corrected non-deadlocking model of the vending machine in [17], with again a large state space, the error occurs if the system runs long enough: it is not very critical in which order operations are performed, as long as the system is running long enough. This explains why for this model breadth-first was performing badly, as it was not generating traces of the system which were long enough to detect the error.

In order to detect both types of errors with a single model checking algorithm, PROB has been using a mixed depth-first and breadth-first search [20]. More precisely, at every step of the model checking, PROB randomly chooses between a depth-first and a breadth-first step.

In summary, the motivation behind PROB's heuristic is that many errors in software models fall into one of the following two categories:

- Some errors are due to an error in a particular operation of the system; hence it makes sense to perform some breadth-first exploration to exercise all the available functionality. In the early development stages of a system model, this kind of error is very common.
- Some errors happen when the system runs for a long time; here it is often not so important which path is chosen, as long as the system is running long enough. An example of such an error is when a system fails to recover resources which are no longer used, hence leading to a deadlock in the long run.

Thus, if the state space is very large, depth-first search can perform very badly as it fails to systematically test combinations of the various operations of the system. Even partial order reduction and bitstate hashing often do not help. Similarly, breadth-first search can perform badly, failing to locate errors that require the system to run for very long. We have argued that PROB's combined depth-first breadth-first search with a random component does not have these pitfalls. In the next section, we will validate this claim empirically.

3 Depth-First versus Breadth-First: An Empirical Evaluation

We report on experiments conducted with PROB using pure depth-first, pure breadth-first, as well as the default mixed depth-first/breadth-first approach of PROB. We also investigate several variations of the mixed approach, varying the probability with which a depth-first step is conducted.

3.1 The Models

We have chosen a variety of case studies for evaluating the effectiveness the various model checking techniques. All models are either classical B models or Rodin Event-B models. We have included several industrial specifications (some stemming from various EU projects, such as Rodin and Deploy¹), as well as academic specifications of various intricate algorithms. There are a few artificial benchmarks as well, testing specific aspects of the model checking algorithm. We have also included some classical puzzles as well, in particular to test directed model checking.

The case studies have been partitioned into four classes:

1. Models with invariant violations,
2. Models with deadlocks,
3. Models with no errors (i.e., no deadlocks or invariant violations), but where a particular goal predicate is to be found. Indeed, in PROB the user can define a particular goal predicate and ask the model checker to find states which make the predicate true. The main difference with point 1 is that the goals are often much more precise (sometimes a concrete particular state) than the invariant violations.
4. Models with no errors, and where the full state space needs to be explored.

A description of the models can be found in the extended version of the paper [18]. Along with the extended version of the paper [18] we have also included the publicly available models.

3.2 The Results

The results are summarised in Tables 11-6 in the Appendix A. Due to space restrictions, we have not included the times for the models without errors; this information can be found in [18]. Relative times are computed with PROB using a mixed depth-first/breadth-first strategy with one-third probability of going depth-first. We call this the reference settings of PROB (prior to the publication of this paper this used to be the default setting; more on that below). The experiments were run on a MacBook Pro with a 3.06 GHz Core2 Duo processor, and PROB 1.3.2 compiled with SICStus Prolog 4.1.2.

¹ EU funded FP7 research project 214158: DEPLOY (Industrial deployment of advanced system engineering methods for high productivity and dependability).

Pure Depth-First. In a considerable number of cases pure depth-first is the fastest method, e.g., for the Peterson_err, Abrial_Earely3_v5, Alstom_axl3, and BlocksWorld benchmarks.

However, we can see in Table 1 that for some models Depth-First fares very badly:

- In Alstom_ex7 in Table 2, pure depth-first search even fails to find the deadlock when given an hour of cputime. This real-life example thus supports our claim from [17] and subsection 2 that when state space is too large to examine fully, depth-first will sometimes not find a counter example. This is actually a quite common case for industrial models: they are typically (at least before abstraction) too large to handle fully.
- Another similar example is Abrial_Press_m13 in Table 3, where pure depth-first is about 900 times slower than PROB in the reference settings.
- Another bad example is Puzzle8, where depth-first is more than 7 times slower or Simpson4Slot where it is 163 times slower than PROB in the reference settings. Finally, for the artificially constructed BFTest, depth-first search fails to find the invariant violation.

For finding goals, the geometric mean of the relative runtimes was 0.92, i.e., slightly better than the reference setting. Overall, pure depth-first seemed to fare best for the deadlocking models with a geometric mean of 0.43. For finding invariant violations, however, the geometric mean was 1.03, i.e., slightly worse than the reference setting.

The bad performance in the Huffman benchmark is actually not relevant: here not all states were evaluated. As such, the time to examine 10,000 states was measured. The pure-depth first search here encountered more complicated states, than the other approaches, explaining the additional time required for model checking.

In conclusion, the performance of pure depth-first alone can vary quite dramatically, from very good to very bad. A such, pure depth first search is not a good choice as a default setting of PROB. Note, however, that we allow the user to override the default setting and put PROB into pure depth-first mode.

Pure Breadth-First. In most cases pure Breadth-First is worse than the reference setting; in some cases considerably so. The geometric mean was always above 1, i.e., worse than the reference setting.

For Alstom_ex7 pure breadth-first also fails to find the deadlock.

Peterson_err in Table 1 gives a similar picture, Breadth-First being 134 times slower than DF and 11 times slower than the reference settings of PROB. Other examples where BF is not so good: Abrial_Early3_v5, DiningPhil, SystemOnChip_Router, Wegenetz.

There are some more examples where it performs considerably better than pure depth-first: Puzzle8, Simpson4Slot, Abrial_Press_m13 and the “artificial” benchmarks BFTest and DFTest2.

In conclusion, breadth-first on its own is not appropriate, except in special circumstances. Note, a user can set PROB into breadth-first mode, but the default is another setting (see below).

Mixed Depth-first/Breadth-first. The motivation behind PROB’s mixed depth-first/breadth-first heuristic is that many errors in software models fall into one of the following two categories:

- Some errors are due to an error in a particular operation of the system; hence it makes sense to perform some breadth-first exploration to exercise all the available functionality. In the early development stages of a model, this kind of error is very common.
- Some errors happen when the system runs for a long time; here it is often not so important which path is chosen, as long as the system is running long enough. An example of such an error is when a system fails to recover resources which are no longer used, hence leading to a deadlock in the long run.

An interesting real-life benchmark is `Alstom_ex7`: here both pure depth-first and pure breadth-first fail to find the deadlock. However, the mixed strategy finds the deadlock.

We have experimented with four different versions of the mixed strategy: DF75, DF50, DF33, DF25. The reference setting was DF33, where there is a 33 % chance of going depth-first at each step. Best overall geometric mean is obtained when using DF50 (which is now the new default setting of PROB).

In summary, let us look at the radar plots in Table 7 in Appendix A, where we summarise the results for pure depth-first, pure breadth-first, the old reference setting and the new one. We can clearly see the quite erratic performance of pure depth-first (relative to the reference setting), and the less erratic but usually worse performance of pure breadth-first. We can also see that the new reference setting usually lies within the reference setting circle, smoothing out most of the (bad) erratic behaviour of the pure depth-first approach.

4 Evaluating Directed Model Checking

Directed Model Checking uses additional information about the model or the destination state in form of a heuristic that guides the model checker towards a target state. This additional information can be collected using for instance static analysis or it can be given by the modeler.

Currently the state space of PROB is stored as a Prolog fact database. Every state can be quickly accessed using its ID or using the hash-value of its state. The model checker also maintains a pending list of open states, using two additional Prolog predicates: retracting a Prolog fact from one of the predicates yields the most recently added open state (for depth-first traversal) and retracting from

the other predicate yields the oldest open state (for breadth-first traversal). This approach allowed us to implement a mixed depth-first / breadth-first approach by randomly selecting either an element from the front or the end of the pending list of open states.

We have implemented a priority queue in C++ using the STL (Standard Template Library) `multimap` data structure. One can thus efficiently add new open states with a particular weight, and then either chose the state with the lowest or highest weight.

We now describe the various heuristic functions we have investigated, as well as the result of the empirical investigation. In this paper we evaluate some strategies to assign weights to newly encountered states. In particular we describe a random search, a search based on the number of successor states, a search based on the (term) size of the state as well as some custom heuristic functions written by the modeler for a particular model. The latter approach is used for models where a specific goal was known, e.g., puzzles.

Random Hash. The idea is simply to use the hash value of a state as the weight for the priority queue. The hope is that the hash value distributes uniformly, i.e., that this would provide a good way to randomize the treatment of pending states. The hash value is computed anyway for every state, using SICStus Prolog `term_hash` predicate.

The purpose was to use this heuristic as a base-line: a heuristic that is worse or not markedly better than this one is not worth the effort. We also want to compare this heuristic with the mixed depth-first/breadth-first approach from Section 3 and see whether there any notable differences. Indeed, the mixed depth-first/breadth-first search does not randomize the order of the states in the list, and this could have an influence on the model checking performance.

Results. For finding deadlocks (Table 5) and goals (Table 6) the random hash heuristic is markedly better than the reference settings of PROB (except for the Bosch cruise control model; but runtimes there are very low anyway). For finding invariant violations, however, (Table 4) it is worse (its geometric mean is greater than 1 (1.07) and in two examples it is markedly worse).

Overall, it seems to perform slightly better than our mixed DF/BF search.

We have also experimented with truly random approach, where we use a random number generator rather than the hash value for the priority. The results are rather similar, except for `Alstom_ex7` where it systematically outperforms Random Hash.

Out Degree. The idea is to use the out degree of a state as priority, i.e., the number of outgoing transitions. The motivation is that if we have found a state with an out degree of 0, i.e., the highest priority, we have found a deadlock. Intuitively, the less transitions are enabled for a state, the closer we may be to a deadlock. In the implementation we actually do not know the out degree of a state until it has been processed. Hence, we use the out degree of the (first) predecessor state for the priority.

Results. Indeed, for finding deadlocks this heuristic obtained the best geometric mean of 0.5. So, this simple heuristic works surprisingly well. For finding goals, this heuristic still obtains geometric mean of 0.63, but it is worse than the random hash function. For finding invariant violations it does not work at all; its geometric mean is 1.56.

A further refinement of this heuristic is to combine the out degree with the random hash heuristic, i.e., if two states have the same out degree (which can happen quite often) we use the hash value as heuristic to avoid a degeneration into depth-first search. This refinement leads to a further performance improvement for deadlock finding (geometric mean of 0.34 compared to 0.50), and for goal finding. But it is markedly worse for invariant violation finding.

In conclusion, the out degree heuristic, especially when combined with random hash, works surprisingly well for its intended purpose of finding deadlocks. In future work, we plan to further refine this approach, by using a static flow analysis to guide model checker into deadlocks and/or particular enablings for events.

Term Size. The idea of this heuristic is to use the term size of the state (i.e., the number of constant and function symbols appearing in its representation) as priority. The motivation for this heuristic is that the larger the state is, the more complicated it will be to process (for checking invariants and computing outgoing transitions). Hence, the idea is to process simpler states first, in an attempt to maximise the number of states processed per time unit.

Results. For finding goals this heuristic has a geometric mean of 0.85, i.e., it is better than the reference setting of PROB, but worse than random hash. For deadlock and invariant checking, it also performs worse than random hash. In summary, this heuristic does not seem worth pursuing further.

Effectiveness of Custom Heuristic Function: In order to experiment easily with other heuristic functions, we have added the possibility for the user to define a custom heuristic function for a B model. Basically, this function can be introduced in the DEFINITIONS part of a B machine by defining HEURISTIC_FUNCTION. PROB now evaluates the expression HEURISTIC_FUNCTION in every state, and uses its value as the priority of the state. Note, the expression must return an integer value. For the BlocksWorld benchmark, we have written the following custom heuristic function:

```

ongoyal == {a|->b, b|->c, c|->d, d|->e};
DIFF(A,TARGET) == (card(A-TARGET) - card(TARGET /\ A));
HEURISTIC_FUNCTION == DIFF(on,ongoyal);

```

Note the machine has a variable `on` is of type `Objects +-> Objects` and the GOAL for the model checker is to find a state where `on = ongoal` is true.

In the benchmarks, we have mainly written heuristic functions which estimate the distance between a target goal state and the current state. In future,

we plan to derive the definition of those heuristic functions automatically. A simple distance heuristic can be derived if the goal of the model checking is to find specific values for certain variables of the machine (such as `on = ongoal`). Basically, for current state $s = \langle s_1, \dots, s_n \rangle$ and a target state $t = \langle t_1, \dots, t_n \rangle$ we use as heuristic $h(s) = \sum_{1 \leq i \leq n} \Delta(s_i, t_i)$ where

- $\Delta(x, target) = abs(x - target)$ if x integer
- $\Delta(x, target) = card(x - TARGET) - card(TARGET \cap A)$ if x a set
- $\Delta((x, y), (t1, t2)) = \Delta(x, t1) + \Delta(y, t2)$ for pairs,
- in all other cases: $\Delta(x, target) = 0$ if $x = target$ and 1 otherwise

If the value of a particular variable is not relevant, then we simply set $\Delta(s_i, t_i) = 0$ for that variable.

This defines a kind of Hamming distance for B states. We have applied this (manually) in the BlocksWorld example above.

We have only evaluated this approach for finding goals. Here, it obtained the best overall geometric mean of 0.34. For Puzzle8 and Abrial_press_m13, this approach yielded by far the best solution. For RussianPostal, TrainTorch, Blocksworld, Abrial_Queue_m1 it obtains the best result. There was one experiment where it is markedly worse than PROB in the reference settings: SystemOnChip_Router. Here the heuristic did not pay off at all. Indeed, here the last event changes all of the four variables, relevant for the model checking GOAL, in one step. This only confirms the fact that we are working with *heuristic* functions, which are not guaranteed to always improve the performance.

Summary of the Directed Model Checking Experiments: We have summarised the main findings of our experiments in Table 8 in Appendix A. We can conclude that:

- for invariant checking, the random hash heuristic fared best.² This seems to indicate that it is maybe useful to combine some more random component into the depth-first/breadth-first techniques of Section 3, e.g., to also randomly permute the operation order. Indeed, the approaches from Section 3 always process the operations in the same order, and do not shuffle the states inside the pending list.
- for deadlock checking, the out-degree-hash heuristic is the best. It should provide a good basis for further improved deadlock checking techniques.
- for goal finding, a custom heuristic function provides (except in one case) by far the best result. The next step is to derive those heuristic functions automatically.

The out-degree-hash heuristic also provides reasonably good performance (its geometric mean is 0.41, which is better than the best mixed-depth-first one of 0.57 for DF75).

² However, note that DF50 had an overall geometric mean of 0.58, and was hence better overall than random hash.

5 Future and Related Work and Conclusion

Regarding directed model checking using heuristics there are a number of other approaches such as a directed extension [10] for Java PathFinder, a tool to modelcheck Java bytecode in order to find deadlocks or problems with null pointers. Edelkamp et al. describe in [7,6,5] various methods to do directed searches for counterexamples to LTL properties within SPIN. A way to use abstraction in order to direct a model checker is described in [4].

Our experiments have confirmed the conjecture of [17], namely that a mixed depth-first/breadth-first strategy for model checking is much more robust than either pure depth-first or breadth-first search. Of particular interest is one industrial model (from Alstom), where neither pure depth-first nor pure breadth-first was capable of detecting the deadlock. We have presented a new model checking algorithm for PROB, which stores the pending list of states in a priority queue. We have presented several heuristic functions and have evaluated them on a wide variety of B models, including several industrial case studies. The experiments have shown that directed model checking can provide a considerable performance improvement. We have shown how one technique, combining the out-degree with a random component, performs very well for finding deadlocks. An adaption of the hamming distance for B states has proven to be very effective in guiding the model checker towards specific goal predicates.

In the future we want to develop more intelligent heuristic functions to guide PROB, e.g., using information we get from an automatic flow analyzer. We currently investigate a method to extract information about the control flow of software systems from Event-B models using a theorem prover. We hope that flow analysis guided model checking will further improve upon the out-degree heuristic for finding deadlocks and would also be helpful to find traces to states where a particular event is enabled. The latter is particularly interesting for test-case generation.

Acknowledgements. We would like to thank the SBMF reviewers for their feedback and many useful suggestions. We are also grateful to the various industrial partners for giving us access to their B models. This research is being carried out as part of the DFG funded research project GEPAVAS and the EU funded FP7 research project 214158: DEPLOY (Industrial deployment of advanced system engineering methods for high productivity and dependability).

References

1. Abrial, J.-R.: The B-Book. Cambridge University Press, Cambridge (1996)
2. Ben-Ari, M.: Principles of the Spin Model Checker. Springer, Heidelberg (2008)
3. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. *Information and Computation* 98(2), 142–170 (1992)
4. Dräger, K., Finkbeiner, B., Podelski, A.: Directed model checking with distance-preserving abstractions. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 19–34. Springer, Heidelberg (2006)

5. Edelkamp, S., Jabbar, S.: Large-scale directed model checking LTL. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 1–18. Springer, Heidelberg (2006)
6. Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Directed explicit-state model checking in the validation of communication protocols. *STTT* 5(2-3), 247–267 (2004)
7. Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Partial-order reduction and trail improvement in directed model checking. *STTT* 6(4), 277–301 (2004)
8. Fleming, P.J., Wallace, J.J.: How not to lie with statistics: the correct way to summarize benchmark results. *ACM Commun.* 29(3), 218–221 (1986)
9. Formal Systems (Europe) Ltd. Failures-Divergence Refinement — FDR2 User Manual (version 2.8.2)
10. Groce, A., Visser, W.: Heuristic model checking for Java programs. In: Bosnacki, D., Leue, S. (eds.) SPIN 2002. LNCS, vol. 2318, pp. 242–245. Springer, Heidelberg (2002)
11. Holzmann, G.J.: The model checker Spin. *IEEE Trans. Software Eng.* 23(5), 279–295 (1997)
12. Holzmann, G.J.: An analysis of bitstate hashing. *Formal Methods in System Design* 13(3), 289–307 (1998)
13. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, Reading (2004)
14. Holzmann, G.J., Peled, D.: An improvement in formal verification. In: Hogrefe, D., Leue, S. (eds.) FORTE. IFIP Conference Proceedings, vol. 6, pp. 197–211. Chapman and Hall, Boca Raton (1994)
15. Hörne, T., van der Poll, J.A.: Planning as model checking: the performance of ProB vs NuSMV. In: Botha, R., Cilliers, C. (eds.) SAICSIT Conf. ACM International Conference Proceeding Series, vol. 338, pp. 114–123. ACM, New York (2008)
16. Jackson, D.: Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology* 11, 256–290 (2002)
17. Leuschel, M.: The high road to formal validation. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 4–23. Springer, Heidelberg (2008)
18. Leuschel, M., Bendisposto, J.: Directed model checking for B: An evaluation and new techniques. Technical report, STUPS, Universität Düsseldorf (September 2010), http://www.stups.uni-duesseldorf.de/publications_detail.php?id=312
19. Leuschel, M., Butler, M.: ProB: A model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
20. Leuschel, M., Butler, M.J.: ProB: an automated analysis toolset for the B method. *STTT* 10(2), 185–203 (2008)
21. McMillan, K.L.: Symbolic Model Checking. PhD thesis, Boston (1993)
22. Peled, D.: Combining partial order reductions with on-the-fly model-checking. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 377–390. Springer, Heidelberg (1994)
23. Samia, M., Wiegard, H., Bendisposto, J., Leuschel, M.: High-Level versus Low-Level Specifications: Comparing B with Promela and ProB with Spin. In: Attiogbe, Mery (eds.) Proceedings TFM-B 2009, pp. 49–61. APCB (June 2009)
24. Wiegard, H.: A comparison of the model checker ProB with Spin. Master’s thesis, Institut für Informatik, Universität Düsseldorf, Bachelor’s thesis (2008)
25. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA⁺ specifications. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 54–66. Springer, Heidelberg (1999)

A Experimental Results

Note: we use geometric mean [8] of the relative runtimes, as the arithmetic mean is useless for normalised results. Of course, the geometric mean itself should also be taken with a grain of salt (various articles also attack its usefulness). Indeed, without knowing how representative the chosen benchmarks are for the overall population of B specifications, we can conclude little.

Thus, we also provide all numbers in the tables below, so that minimum, maximum relative runtimes can be seen, as well as the absolute runtime of the reference benchmark. Indeed, the relative runtimes are less reliable, when the absolute runtime of the reference benchmark is already very low. When a given technique failed to locate the invariant violation (respectively deadlock or target goal), then we have marked the time with two asterisks (**). The tables for the models without errors where the full state space has to be explored can be found in [18].

Table 1. Relative times for checking models with invariant violations (DF/BF)

Invariant Benchmark	DF	DF75	DF50	DF33 (abs+rel)	DF25 Rel	BF
SchedulerErr	0.33	0.33	0.33	30 ms 1.00	1.00	2.67
Simpson4Slot	163.33	0.22	0.67	90 ms 1.00	0.78	2.11
Peterson_err	0.08	0.08	0.22	360 ms 1.00	3.06	11.17
TravelAgency	0.16	0.27	0.10	630 ms 1.00	0.78	2.43
SecureBldg_M21_err3	0.50	0.50	1.00	20 ms 1.00	1.00	1.00
Abrial_Press_m2_err	0.26	3.38	0.34	880 ms 1.00	1.81	1.26
SAP_M.Partner	0.58	1.08	1.00	120 ms 1.00	0.92	0.83
NastyVending	0.02	0.08	8.00	130 ms 1.00	2.85	1.00
NeedhamSchroeder	1.28	1.52	0.95	22620 ms 1.00	0.70	1.37
Houseset	0.05	0.06	0.22	2610 ms 1.00	2.66	** 336.27
BFTest	** 15000.00	11592.75	0.75	80 ms 1.00	1.00	0.88
DFTest1	0.20	0.35	0.71	2360 ms 1.00	1.01	1.02
DFTest2	7.86	0.50	0.60	2930 ms 1.00	0.99	1.00
GEOMEAN	1.03	0.78	0.58	394 ms 1.00	1.24	2.35

Table 2. Relative times for checking models with deadlocks (DF/BF Analysis)

Deadlock Benchmark	DF	DF75	DF50	DF33 (abs+rel)	DF25 Rel	BF
Abrial_Earley3_v3	0.33	0.44	0.93	270 ms 1.00	1.19	1.19
Abrial_Earley3_v5	0.13	0.32	0.75	4320 ms 1.00	2.18	7.95
Abrial_Earley4_v3	0.89	0.89	1.00	90 ms 1.00	1.00	1.00
Alstom_axl3	0.10	0.17	0.20	51270 ms 1.00	3.51	14.61
Alstom_ex7	** 4.20	0.23	0.35	856320 ms 1.00	** 1.21	** 3.00
Bosch_CrsCtl	1.00	1.00	1.00	3 ms 1.00	4.00	4.00
SAP_MChoreography	0.50	0.50	0.50	20 ms 1.00	1.00	1.00
DiningPhil	0.13	0.26	0.36	1690 ms 1.00	2.49	6.20
CXCC0	0.50	1.00	1.00	10 ms 1.00	2.00	2.00
GEOMEAN	0.43	0.44	0.59	540 ms 1.00	1.82	3.02
AVG	0.00	0.00	0.00	0 ms 0.00	0.00	0.00

Table 3. Relative times for checking models with goals to be found (DF/BF Analysis)

Goal Benchmark	DF	DF75	DF50	DF33 (abs+rel)	DF25 Rel	BF
RussianPostal	0.95	0.14	0.73	220 ms 1.00	1.05	1.50
TrainTorch	1.14	1.17	0.69	350 ms 1.00	1.06	0.94
BlocksWorld	0.07	0.25	1.16	440 ms 1.00	1.18	1.07
Farmer	0.50	1.00	0.50	20 ms 1.00	1.00	1.00
Hanoi	0.54	0.48	1.00	500 ms 1.00	0.84	0.90
Puzzle8	7.56	2.86	0.40	59060 ms 1.00	0.11	0.71
RushHour	0.41	0.66	0.90	127020 ms 1.00	1.09	1.11
Abrial_Press_m13	899.78	1.64	1.26	800 ms 1.00	0.66	2.23
Abrial_Queue_m1	1.60	3.00	1.00	50 ms 1.00	1.60	1.40
SystemOnChip_Router	0.05	0.14	0.08	2050 ms 1.00	1.04	1.45
Wegenetz	0.08	0.08	0.25	120 ms 1.00	0.17	2.58
GEOMEAN	0.92	0.57	0.58	715 ms 1.00	0.71	1.26

Table 4. Relative times for checking models with invariant violations (Heuristics)

Invariant Benchmarks	DF33 (abs+rel)	HashRand	OutDegree	OutDegHash	TermSize
SchedulerErr	30 ms 1.00	0.33	3.67	10.67	2.67
Simpson4Slot	90 ms 1.00	0.89	2.22	0.78	2.22
Peterson_err	360 ms 1.00	0.75	11.25	1.42	9.14
TravelAgency	630 ms 1.00	0.52	1.02	9.98	0.49
SecureBldg_M21_err3	20 ms 1.00	0.50	1.00	0.50	0.50
Abrial_Press_m2_err	880 ms 1.00	1.45	3.38	3.19	1.25
SAP_MPartner	120 ms 1.00	1.17	0.75	0.33	1.00
NeedhamSchroeder	22620 ms 1.00	1.40	**48.51	41.58	** 46.06
Houseset	2610 ms 1.00	0.08	** 333.53	0.16	** 338.61
BFTest	80 ms 1.00	16.00	0.88	73.00	0.88
DFTest1	2360 ms 1.00	0.64	1.02	0.69	1.02
DFTest2	2930 ms 1.00	0.53	1.00	0.57	1.66
GEOMEAN	432 ms 1.00	0.79	2.44	1.54	1.93

Table 5. Relative times for checking models with deadlocks (Heuristics Analysis)

Deadlock Benchmarks	DF33 (abs+rel)	HashRand	OutDegree	OutDegHash	TermSize
Abrial_Earley3_v3	270 ms 1.00	0.74	1.22	0.70	1.22
Abrial_Earley3_v5	4320 ms 1.00	0.41	5.88	0.16	7.95
Abrial_Earley4_v3	90 ms 1.00	0.89	1.00	0.89	1.00
Alstom_axl3	51270 ms 1.00	0.82	0.09	0.16	0.08
Alstom_ex7	856320 ms 1.00	0.55	** 1.91	1.10	**1.57
Bosch_CrsCtl	3 ms 1.00	8.00	1.00	1.00	4.00
SAP_MChoreography	20 ms 1.00	0.50	0.50	0.50	1.00
DiningPhil	1690 ms 1.00	1.16	0.05	0.03	1.59
CXCC0	10 ms 1.00	0.25	0.25	0.25	0.25
GEOMEAN	432 ms 1.00	0.80	0.58	0.34	1.07

Table 6. Relative times for checking models with goals to be found (Heuristics)

Goal Benchmarks	DF33 (abs+rel)	HashRand	OutDegree	OutDeg-Hash	TermSize	CUSTOM
RussianPostal	220 ms 1	0.45	0.77	0.36	1.18	0.45
TrainTorch	350 ms 1	0.97	0.26	1.14	0.20	0.20
BlocksWorld	440 ms 1	1.16	0.07	0.07	1.20	0.02
Farmer	20 ms 1	1.00	1.00	0.50	1.00	1.00
Hanoi	500 ms 1	0.52	0.92	0.52	0.90	0.34
Puzzle8	59060 ms 1	20.54	1.31	2.69	0.71	0.03
RushHour	127020 ms 1	0.42	0.60	0.56	1.12	0.79
Abrial_Press_m13	800 ms 1	0.49	2.36	2.21	2.48	0.20
Abrial_Queue_m1	50 ms 1	0.40	16.60	0.60	2.80	0.40
SystemOnChip...	2050 ms 1	0.10	0.05	0.04	1.50	72.42
Wegenetz	120 ms 1	0.17	0.33	0.08	0.08	0.08
GEOMEAN	715 ms 1	0.64	0.63	0.41	0.85	0.34

Table 7. Radar plots for invariant, deadlock and goal checking (DF/BF)

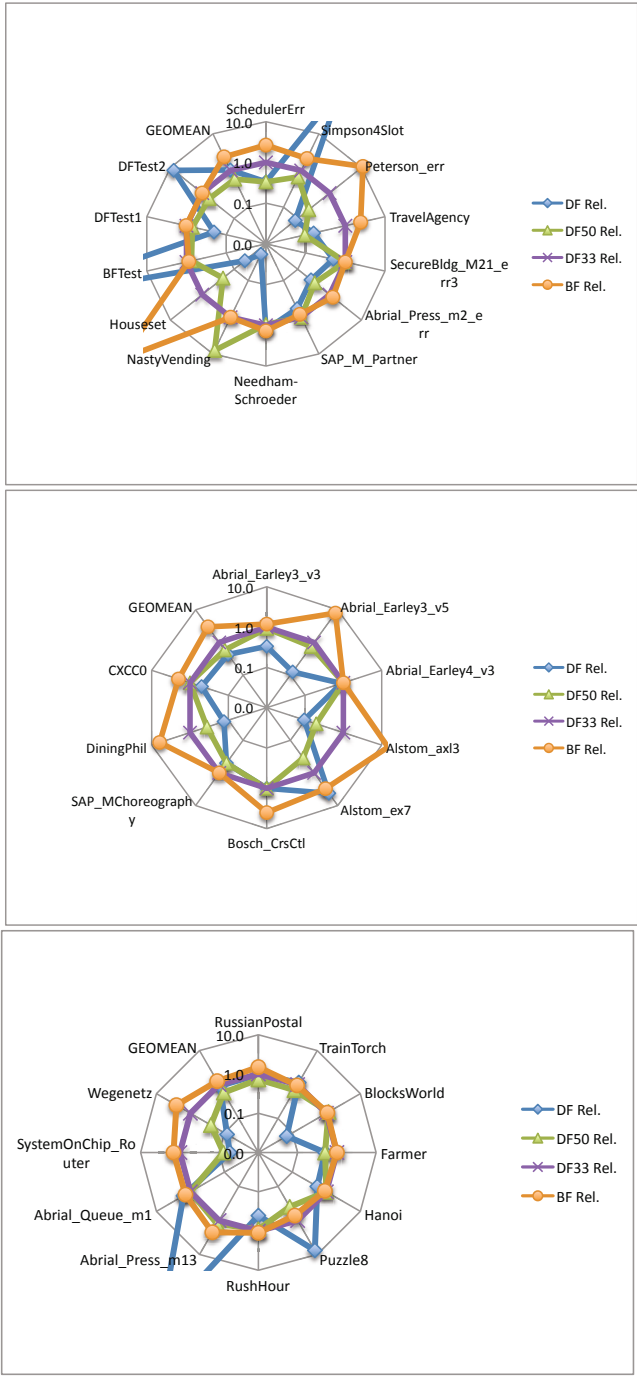
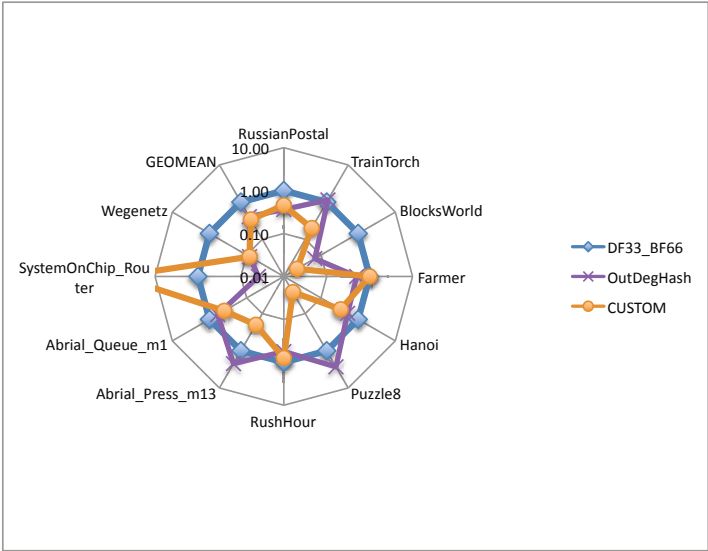
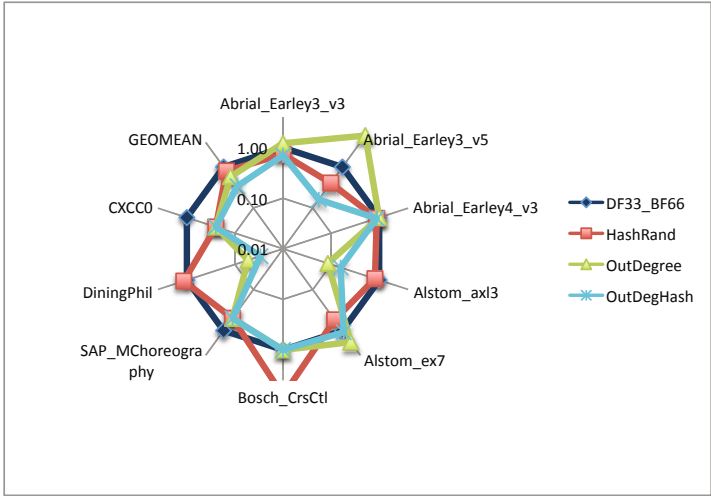


Table 8. Radar plots for deadlock and goal checking (Heuristics)



Midlet Navigation Graphs in JML

Wojciech Mostowski and Erik Poll

Radboud University Nijmegen
Digital Security Group
woj@cs.ru.nl, erikpoll@cs.ru.nl

Abstract. In the context of the EU project Mobius on Proof Carrying Code for Java programs (midlets) on mobile devices, we present a way to express midlet navigation graphs in JML. Such navigation graphs express certain security policies for a midlet. The resulting JML specifications can be automatically checked with the static checker ESC/Java2. Our work was guided by a realistically sized case study developed as demonstrator in the project. We discuss practical difficulties with creating efficient and meaningful JML specifications for automatic verification with a lightweight verification tool such as ESC/Java2, and the potential use of these specifications for PCC.

1 Introduction

Midlet navigation graph provide a way to specify security properties for Java MIDP (Mobile Information Device Profile) applications, so-called *midlets*. Based on a simpler notion of a flow graph, prescribed by the Unified Testing Criteria (UTC) [20] to test midlets in Java Verified scheme [4], Crégut proposed the notion of navigation graphs [5] as a high-level specification formalism to describe the behaviour of Java mobile phone applications (in most cases MIDP devices are in fact mobile phones).

Essentially, a navigation graph is a graph, or finite automaton, which describes the ways in which an application may navigate through various screens of the user interface, in interaction with the user and the network. Each node in the graph represents a different screen that is displayed, e.g. a warning message for which the user has to press ‘OK’ or ‘Cancel’, or a menu with options for the user to choose from. The arrows between nodes represent transitions the application can make, often in response to some user action. The graph can be augmented with information about sensitive midlet actions, e.g. sending an SMS or engaging in other GSM network activity. The navigation graph then gives a high-level specification of which potentially dangerous things a given midlet does, and under which circumstances.

In [5] Crégut gives a formal description of navigation graphs and their semantics in terms of an operational semantics of Java bytecode, more specifically the Bicolano semantics [18]. He also presents an algorithm to extract a navigation

¹ <http://www.javaverified.com>

graph from bytecode. In this paper we present a way to express the semantics of navigation graphs in terms of a specification at source code level. The formal specification language we use for this is Java Modelling Language (JML) [14].

Our work was guided by one of the Mobius case studies, a quiz game midlet developed by industrial partner TLS² [15]. This is the biggest case study of the project and it exhibited some problems during the JML annotation process and also during verification with the extended static checker for Java, ESC/Java2 [11]. We will discuss the problems we encountered with developing the JML specification and verifying it in detail.

The rest of this paper is organised as follows. Sect. 2 gives an introduction to the MIDP application structure. Sect. 3 describes midlet navigation graphs in more detail. Sect. 4 discusses the translation of the midlet navigation graphs in JML based on the Mobius case study. Finally, Sect. 6 summarises our results and discusses the potential for PCC.

2 MIDP Infrastructure

The notion of midlet navigation graphs relies on the infrastructure of the Java2 Micro Edition platform (J2ME)³ for small devices, such as mobile phones or PDAs. The main building blocks of the J2ME platform of interest are the Mobile Information Device Profile (MIDP) Java API and the Connected Limited Device Configuration (CLDC) Java API. The former API deals with output to the display and input from the user via the keypad of the device. The latter API is mostly responsible for device's communication with the outside world. J2ME is often referred to as MIDP, and CLDC is usually assumed to be part of J2ME/MIDP when mobile phones or PDAs are considered. The applications that run on these devices are called midlets.

GUI. A navigation graph is related to the phone's display and sensitive operations that a midlet can possibly perform. In the following we discuss the relevant parts of the MIDP API. As a presentation aid we use UML.

Every midlet, represented by the `MIDlet` class, has a unique associated `Display` object, which manages the display and the input devices. The static method `Display.getDisplay(MIDlet m)` returns the display associated with a midlet. The various kinds of things that can be displayed on the `Display` object are instances of the subclasses of `Displayable`, shown in Fig. 1. Invoking the method `void setCurrent(Displayable nextDisplayable)` on a `Display` changes what it displays, possibly after a short delay. A `List` presents a list of choices, i.e. a menu, that the user can scroll through and select. A `TextBox` allows the user to enter and edit text. A `Form` presents an arbitrary mixture of items, which can for instance be images or (read-only or editable) text fields. Finally, an `Alert` is a screen that is shown for a short period of time, either until some timeout or a key press. Alerts can also be displayed by invoking the method `void`

² <http://www.tls.pl>

³ <http://java.sun.com/javame/index.jsp>

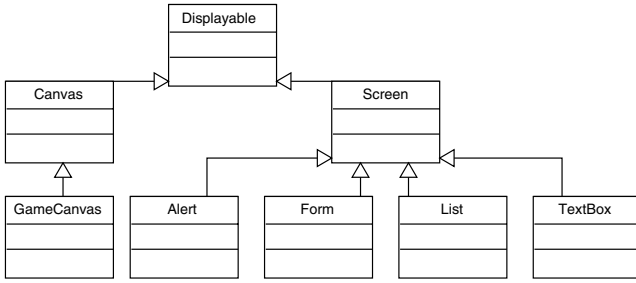


Fig. 1. Class diagram: basic MIDP GUI elements

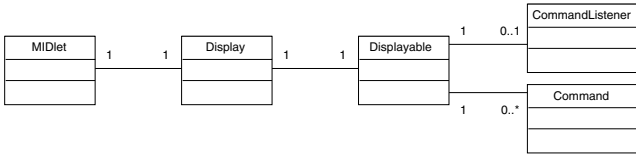


Fig. 2. Class diagram: relation between midlets, displays, and screens

`setCurrent(Alert alert, Displayable nextDisplayable)`, which will display the alert, and then show the `nextDisplayable`.

In its turn, a `Displayable` object may have a `CommandListener` associated with it, which implements a method `void commandAction(Command c, Displayable d)` to handle incoming command events occurring on some `Displayable d`. `Commands` are user actions, such as buttons that the user can select on the screen. Fig. 2 shows the relevant class structure. Note that control passes back and forth between the midlet and the platform. When a midlet calls `setCurrent(...)` to change the display, it hands over control to the platform; when after that a user action occurs, the platform hands back control to the midlet by a call back to `commandAction(...)`. The behaviour of the midlet is determined by: (i) the current `MIDlet` and its `Display`, (ii) the current `Displayable` shown on that `Display`, (iii) the `Commands` that the midlet provides (if any), (iv) the associated `CommandListener` (if any). The display and the midlet should never change. The MIDP platform controls which `Displayable` is shown, and offers a midlet API calls to change it; the midlet is in charge of the `Commands` and `CommandListeners` and their associations to `Displayables`.

Sensitive Operations. The second relevant part of the MIDP infrastructure are the APIs responsible for network communication and personal information management. These operations are possibly security-sensitive. An unwanted or uncontrolled network communication may result in (a) sensitive data being sent out from the phone, or (b) unwanted network usage charges. Access to personal information (e.g. the phone book) may result in unwanted information leakage.

The high level API structure for network communication is very simple. In principle it only involves the `Connector` class that provides static methods for

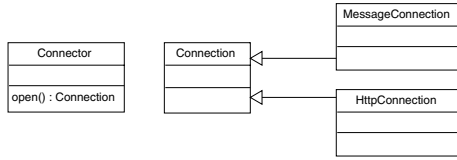


Fig. 3. Class diagram: network connection related API

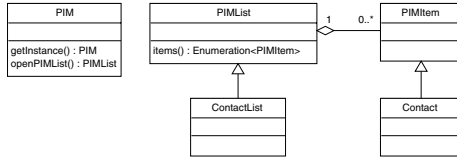


Fig. 4. Class diagram: personal information related API

establishing different kinds of network connections (SMS, Internet), and a few classes that encapsulate these different connections, like `MessageConnection` or `HttpConnection`. Fig. 3 gives a simplified view.

For personal information management (PIM) there is one utility class `PIM` that provides methods for accessing the phone book (contact list), and a few classes that represent a single contact or the whole contact list (Fig. 4).

3 Navigation Graphs

In [5] two formalisations are given to deal with midlet navigation graphs. The first formalisation deals with the MIDP GUI structure. We described it intuitively using UML. In [5] a more detailed semantics is given in terms of the Bicolano semantics [18] of Java bytecode; there the formalisation of the GUI is needed to develop the algorithm to generate navigation graphs out of bytecode. For our purposes the lightweight UML representation of the GUI is sufficient to represent the GUI behaviour in JML by annotating the parts of the MIDP API dealing with the GUI. Although we do not use the detailed formalisation of the GUI from [5], there is a close correspondence between that formalisation and our JML representation of the graphs. E.g., our $1 - 0..1$ relation between `Displayables` and `CommandListeners` is the relation $g.list$ in [5], where g denotes the state of the GUI and the whole relation maps `CommandListeners` to `Displayables`. Similarly, the relation $g.coms$ in [5] corresponds to our $1 - 0..*$ mapping between `Displayables` and `Commands`.

The second part of the formalisation in [5] gives a formal definition of a midlet navigation graph itself. Putting aside the complex notation, a midlet navigation graph is essentially an oriented multigraph. The nodes of this graph are possible midlet states, i.e. different application screens. The arrows of the graph are

transitions between the screens caused by user actions, i.e. **Commands**. Finally, in [5] the arrows (transitions) also have interpretations, as they indicate which sensitive operations that may be performed during a given transition.

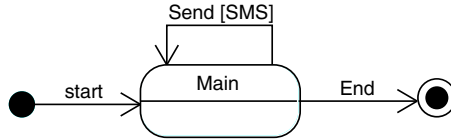


Fig. 5. Statechart diagram: a simple midlet navigation graph

Such a notion of a graph can also be easily represented by a UML state chart diagram. The states of the diagram represent application screens, the arrows represent user commands, and arrow guards can be used to mark sensitive operations. A very simple example is given in Fig. 5. In this example, from the main screen a user can choose the **Send** command, in which case at most one SMS would possibly be sent over the network, or press **End**, in which case the application will simply terminate. Furthermore, using the UML state stereotypes we can indicate additional properties of screens, e.g. whether an alert screen is displayed only for a given period of time indicated by the timeout parameter (and hence performing a transition to another screen without user interaction). This last aspect is not covered in [5].

4 Navigation Graphs in JML

This section gives the semantics of a navigation graph in terms of JML. In other words, we define a mapping from navigation graphs to JML annotations. Our effort is divided into two parts. We start with the first part, which is to specify, in a generic way, the midlet API calls. The API specifications can then be used when specifying a particular midlet behaviour to reflect a given midlet navigation graph in the second part that we discuss later.

4.1 Relevant API Methods

We want our JML specifications to be easy to verify for the verification tools. Hence we only specified those aspects that are needed for navigation graphs, and we avoided constructs that are challenging for verification, as discussed later. Our specifications often use so-called ghost variables [4], which are specification-only variables, to model relevant aspects of the state of the platform (incl. the GUI).

As mentioned before, two aspects of the API need to be specified: the GUI and security-sensitive operations. For the GUI, we need to specify the **Display** class, where a ghost field **current** tracks what is being displayed:

```

public class Display {
    // Display represents the manager of the display.
    // There is exactly one instance of Display per MIDlet

    //@ public non_null ghost MIDlet midlet;
    //@ public non_null ghost Displayable current;
    //@ public non_null ghost Alert preAlert;

    //@ ensures \result != null && \result.midlet == m;
    //@ assignable \nothing;
    public /*@pure@*/ static Display getDisplay(/*@non_null@*/ MIDlet m);

    //@ ensures current == nextDisplayable && preAlert == null;
    //@ assignable current, preAlert;
    public void setCurrent(/*@non_null@*/ Displayable nextDisplayable);

    //@ ensures current == nextDisplayable && preAlert == alert;
    //@ assignable current, preAlert;
    public void setCurrent(/*@non_null@*/ Alert alert,
                           /*@non_null@*/ Displayable nextDisplayable);
}

```

For the second `setCurrent` method we made a practical simplification. This method causes an `alert` screen to be displayed temporarily (either with a time-out or with a confirmation button) before updating the display to show `nextDisplayable`. This could be specified by writing a complex specification that keeps track of the sequence of displayed screens, including these alerts. However, verification would be much more difficult and with little added value. Instead, we introduce a ghost field, `preAlert`, which records that an alert screen is temporarily displayed. So `current` tracks the displayables being shown over time, ignoring temporary `Alert` displayables.

To specify the GUI behaviour, we also have to specify the `Displayable` class that represents particular screens on the display and the `Command` class that represents input events. To simplify things we assume that each `Command` object is bound to only one `Displayable`. Generally, this does not have to be the case (commands can be reused through different displayables). However, in practice most midlets define separate commands for each screen, and requiring it makes verification simpler, as we do not need to use sets (or some representation of sets such as lists) to track the set of displays associated with a command, and then use set theory in verification. As for command listeners, the platform enables only one per screen:

```

public class Command {
    // The (only) Displayable object this command is attached to
    //@ public ghost Displayable displayable;
}

public class Displayable {
    //@ public ghost CommandListener commandListener;
}

```

```

    /**@ ensures cmd.displayable == this; assignable cmd.displayable;
    public void addCommand(/*@non_null@*/ Command cmd);

    /**@ ensures commandListener == l; assignable commandListener;
    public void setCommandListener(/*@non_null@*/ CommandListener l);
}

```

Finally, the specification of `CommandListener` should reflect the MIDP platform guarantees, namely that the current displayable and command are not null, and that the invoked command is in fact associated with the given displayable⁴:

```

public interface CommandListener {
    /**@ requires c.displayable == d;
    /**@ assignable \everything;
    public void commandAction(/*@non_null@*/ Command c,
                              /*@non_null@*/ Displayable d);
}

```

If we trust the platform not to behave abnormally, these assumptions are safe.

For security-sensitive API calls, that may result in say network usage or access to private information, we want our API specification to track the number of invocations. For this we declare suitable static ghost variables to count the number of invocations. This allows us to express restrictions on these numbers in a specification for midlet. For example, for the `open` method of the `Connector` class, which establishes new network or SMS connections, we can specify

```

public class Connector {
    /**@ public static ghost int openCount;
    /**@ ensures Connector.openCount == \old(Connector.openCount) + 1;
    /**@ assignable Connector.openCount;
    public static Connection open(/*@non_null@*/ String name);
}

```

4.2 Midlet Annotations – The Mobius Case Study

We will present the JML annotations for midlets using the Mobius demonstration midlet. The midlet implements a simple mobile phone quiz game. The security sensitive operations of the quiz game are using an HTTP connection to download game questions, sending answers and scores in SMS messages, and also accessing the Personal Information Manager (PIM), i.e. the phone book. Fig. 6 shows the complete navigation graph of this midlet.

The application class structure is as follows. There is a singleton class `QuizMidlet` which is the main application container. Then there are several classes to represent different screens of the game: main menu, options screen, about screen, the main game screen, etc. Most of these classes use the Singleton pattern, meaning there will only ever be a single instance of them. These classes also implement

⁴ The assignable `\everything` clause in the spec means that classes implementing this method are in principle free to have any side-effects.

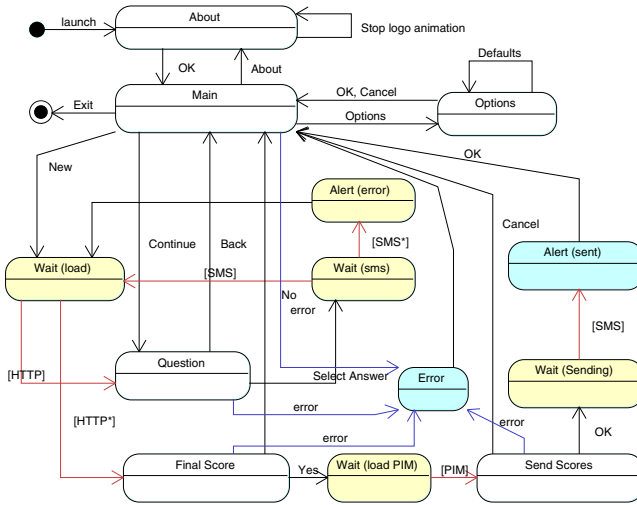


Fig. 6. State diagram: navigation graph for the Mobius game

the `CommandListener` interface to handle user actions. The class `QuizQuestion` encapsulates a single quiz question and a displayable object for this question. Finally, there are two utility classes that handle network connections and PIM access. The whole application consists of 13 small classes.

We start with specifying possible screens and screen transitions of our midlet. As it turns out this is the more difficult part and also one that exhibits the biggest problems with accurate specification of the navigation graph in JML. Later we deal with the calls to sensitive operations.

Screen State. To specify the navigation graph accurately we need to keep track of screen changes in our midlet. For this we use the `current` instance field of the `Display` object associated with our midlet. A suitable invariant enforces the limit on the set of possible screens, as follows:

```
public class QuizMidlet extends MIDlet {
    private /*@ spec_public non_null @*/ Display display;
    private /*@ spec_public non_null @*/ MainMenu mainMenu;
    /*@ invariant display.current == mainMenu.list ||
        display.current == About.about.alert ||
        display.current == Options.opts.form ...; @*/
```

However, we quickly run into problems with completing this invariant, because for every screen – i.e. `Displayable` – the application uses we need some program variable (like `mainMenu.list`) to refer to it. Such variables do not always exist. Some objects of type `Displayable` are created on the fly, and cannot be referred to from the class `QuizMidlet`. For example, this is the case for the `FinalScore` screen in the game, which is created locally in the `MainMenu` class:


```
public void quizFinished() {
    int score = currentGame.getScore(); ...
    FinalScore finalScore = new FinalScore(midlet, score);
    finalScore.show(getDisplayable());
}
```

The `finalScore` object, the screen that displays the score, is only visible locally in the `quizFinished` method and it cannot be referred to in a global midlet invariant. There are other examples of such locally created displayables in the midlet code.

To solve this problem we turn to static ghost variables. Although the problematic displayable objects are created locally, there is only one object of a given kind created and possibly active at a time. So we simply store such locally created displayable object in a static ghost variable. Since we can make the static variable public it will be in scope for all our specifications. The fact that it is static lets us make sure that we keep track of only the most recently (and thus current) created displayable object of a given type, and also that access to this ghost variable is object reference independent. For the `FinalScore` class the relevant annotations are the following:

```
public class FinalScore implements CommandListener {
    //@ public static ghost Alert displayable;
    public void show(Displayable next) { ...
        Alert alert = new Alert("Final Scores");
        //@ set FinalScore.displayable = alert;
        alert.setTimeout(Alert.FOREVER); ...
        midlet.getDisplay().setCurrent(alert);
    }
}
```

Then our invariant can refer to the `FinalScore.displayable` field:

```
/*@ invariant ... display.current == Options.opts.form ||
    display.current == FinalScore.displayable || ... ; @*/
```

However, this solution brings up another problem. The visible state semantics of invariants requires all invariants of all objects to hold in all visible states (i.e. all pre- and post-states of all method calls) during the execution of our midlet. This obviously does not hold in the code above. Our invariant is broken in all the states between the state where `FinalScore.displayable` is set and the state when the display is updated by invoking `setCurrent`. In these intermediate states `display.current` may point to a reference that is not stored in `FinalScore.displayable` anymore.

A (standard) trick we use to solve this problem is introducing a global boolean guard, `Display.displayUpdated`, to switch invariants on and off at appropriate points, as shown below.

```

public class Display {
    /*@ public static ghost boolean displayUpdated;

    /*@ ensures current == nextDisplayable && preAlert == null;
    /*@ ensures Display.displayUpdated;
    /*@ assignable current, preAlert, Display.displayUpdated;
    public void setCurrent(/*@non_null@*/ Displayable nextDisplayable); }

    /*@ invariant Display.displayUpdated ==>
        display.current == Options.opts.form ||
        display.current == FinalScore.displayable || ...; @*/

```

By setting `Display.displayUpdated` to false we can temporarily ‘switch off’ the invariant. The specification of `setCurrent` ensures that the guard is re-established, so that the invariant has to hold after every call to `setCurrent`.

Screen Transitions. The invariant above suffices to limit the set of screens of a given midlet. In the next step we need to specify when and how screen transitions happen. In general, this is a very difficult problem: midlets are concurrent applications and screens can be changed by the J2ME environment at any time without user interaction. A notable example of this is an incoming call on a mobile phone, or simply midlet environment warning screens, e.g. to confirm sensitive operations. Note that we have already skipped such screens in the specifications above, and in the navigation graph too. The non-deterministic character of such screens would make the graph and the specification unnecessarily complex. Furthermore, we are interested in verifying the application itself rather than the whole environment it runs in.

Apart from the cases mentioned above, the midlet screen transitions are triggered by user input and actions. All user actions are handled by different implementations of the `commandAction` method. This is where we add annotations to limit the possible screen changes. The precondition limits the set of commands that can be invoked on this screen, the postcondition describes how the `FinalScore` screen will change after processing the command:

```

/*@ requires c==cmd_yes || c==cmd_no;
/*@ requires next == midlet.mainMenu.list;
/*@ ensures c==cmd_no ==> midlet.display.current == next;
/*@ ensures c==cmd_yes ==>
    midlet.display.current == SendScores.displayable;
/*@ assignable ...;
public void commandAction(Command c, Displayable d) {
    if (c == cmd_no) {
        midlet.getDisplay().setCurrent(next); }
    else if (c == cmd_yes) {
        SendScores sendScores = new SendScores(midlet);
        sendScores.show(score, next); }
}

```

Sensitive Operations. In the last step we limit the sensitive operations our midlet performs. For any method that ultimately calls down to one of these operations we need to add a contract for the associated ghost variable that, as described at the end of Sect. 4.1, tracks the number of invocations.

```
/*@ ensures Connector.openCount == \old(Connector.openCount);
public void commandAction(Command c, Displayable d) { ... }
```

Then we allow the sensitive operations to be performed by the implementations of the `commandAction` that correspond to transitions in the graph:

```
public class SendScores {
  //@ ensures c == cmd_ok ==>
    MessageConnection.smsSent <= \old(MessageConnection.smsSent) + 1;
  //@ assignable MessageConnection.smsSent, ...;
  public void commandAction(Command c, Displayable d) {
    ... else if (c == cmd_ok) {
      WaitAlert.getInstance().show(midlet, Consts.SENDING_RESULT);
      String s = numberField.getString();
      sendSMS(s); ... }
  }
}
```

A similar approach is used to limit access to personal data (e.g. the phone book) in the MIDP environment.

If properties are expressed in postconditions, we have to consider the issue of non-termination. Verification would have to be done using total correctness to ensure that, say, limits on the number of SMS sent are not broken in non-terminating executions of a method. Still, for `commandAction` methods which do not have the `MessageConnection.smsSent` in their assignable clause, the contract does rule out that any SMS are sent in non-terminating executions.

5 Specification and Verification Issues

During the verification of this case study with ESC/Java2 two practical issues surfaced. The first one has to do with singleton pattern classes, the second one with visible state semantics of invariants.

Singleton Objects. Many classes in midlet code study and in the MIDP API follow the singleton pattern [8], i.e. only one instance of these classes is ever created. Knowing that a class is only ever going to have a single instance could in principle simplify reasoning. However, specifying that a class follows the singleton pattern in JML, and verifying it with ESC/Java2, can be a bit clumsy. E.g., specifying that `Options` is a singleton class could be specified by an invariant

```
private /*@ spec_public @*/ static Options instance = null;
/*@ invariant this == instance;
```

saying that all `Options` objects are equal to the static `instance`. However, this invariant is typically too strong, given the visible state semantics of invariants. If the singleton object is created by a static get-method as is done in the code, e.g.

```

/*@ ensures \result != null && \result == instance;
   */
/*@ assignable Options.instance;
   */
public static Options getInstance() {
    if (instance == null) instance = new Options();
    return instance;
}

```

then the constructor call `new Options()` by itself does not establish the invariant, as it will not hold till after the assignment to `instance`.

A possible solution is to make the constructor a helper:

```

public /*@ helper @*/ Options() {...}

```

effectively telling ESC/Java2 to inline calls to the constructor, but it turns out that ESC/Java2 reasons in a rather unpredictable way when a private constructor is declared as `helper`.

In the end we let ESC/Java2 complain about the possibly unsatisfied `this == instance` invariant after a call to `new Options()` in `getInstance()` and suppressed this warning with the `@nowarn` directive.

It would be nice to have a standard, e.g. following ideas from [19], simple way of specifying singleton behaviour in JML, so that no necessary complications and side conditions are introduced during reasoning.

Visible State Semantics of Invariants. JML uses the visible state semantics for object invariants [13]. This means that all invariants of all objects of all types have to hold in all *visible states*, which includes all post-states of constructor calls and all pre- and post-states of method calls⁵.

This semantics makes verification very hard, and non-modular⁶: when verifying a method in one class one should take into consideration breaking invariants of other objects, of any class, that happen to be allocated. To enable modular verification, ESC/Java [7] uses a slightly weaker and potentially unsound semantics. ESC/Java2 [11] uses the same semantics, but now includes features to warn users about potential problems [12,10].

Working on the case study, this generated several false positives, where the code was incorrectly deemed to be correct. The root of the problem was that when checking code that calls API methods, ESC/Java2 will assume that these API methods preserve all invariants. When checking the implementation of these API methods the tool would probably warn that they do not preserve invariants,

⁵ Except those constructors and methods designated as `helper`'s.

⁶ However, the semantics is sound, unlike more simple-minded semantics for object invariants!

but as we are only ever checking the midlet code – i.e. client code of the API – and never implementation of this API, this issue can easily go undetected.

The example below illustrates this:

```
public class APIClass {
  //@ requires array.length == 1;
  //@ ensures array[0] == v; assignable array[0];
  public static void setArray(/*@ non_null @*/ int[] array, int v);
}

public class ClientClass {
  private /*@ non_null @*/ int[] values = {1};
  //@ invariant values[0] == 1 && values.length == 1;

  public void modifyArray() { APIClass.setArray(values, 2); }
}
```

Checking the `modifyArray` method with ESC/Java2 does not show any problems. However, it is clear that a call to `APIClass.setArray` breaks the class invariant for `ClientClass`. However, the tool assumes that the API class will re-establish all invariants.

Running the tool on an implementation of the `setArray` method would probably reveal that the invariant of `ClientClass` might be broken, but typically one treats the API as a black box and one does not look at implementations of it. Unfortunately the inconsistency warning of ESC/Java2 [12,10] does not catch these situations.

Sound and modular verification techniques that cope with class invariants do exist [6], and are for instance used in Spec# [1]. Some verification tools, like KeY [3], allow a very flexible invariant semantics; there it is up to the user to choose which class (or even method) is responsible for a given invariant, but that means soundness is up to the user too.

6 Evaluation and Discussion

An issue often overlooked in research on program verification is coming up with interesting properties to verify. We have shown a way to specify midlet navigation graphs by means of JML annotations. This required some generic annotation of the MIDP API, which provides a ‘ghost state’ to talk about the relevant platform infrastructure – ghost fields that track which `Displayable` is being shown, and hence which `CommandListener` is active, etc. – and contracts for some API calls that describe their effect on this ghost state. An individual midlet can then be annotated to express conformance to a midlet navigation graph, by introducing an invariant that restricts the possible `Displayables` that can be shown, and contracts for `commandAction` methods that specify the screen transitions that are allowed. Additional restrictions on security-sensitive API calls, saying in which states these may occur, can be expressed if API specifications for these methods are added to track their usage.

The translation of a midlet navigation graph to JML, which we did by hand, could be automated, as done for state diagrams by the AutoJML tool [9]. An alternative to coding up the midlet navigation graph in JML pre- and postconditions, as we have done, would be to use special specification constructs for temporal properties [21] or CSP-style constraints on method sequences [16].

Our approach has been shown to work on a non-trivial (albeit still very small) midlet, the Mobius Quiz game demonstrator, which consists of 13 classes and 1350 lines of code, which was then verified using ESC/Java2. Once the midlet navigation graph is expressed by JML annotations, further annotations of the midlet are needed for the verification to go through, e.g. to rule out `NullPointerExceptions`, etc. This further annotation took an effort in the order of days.

As explained in Sect. 4.2, a technical complication is any use of dynamically created `Displayable` objects in midlet code, as additional static (ghost) fields have to be introduced to refer to these objects in specifications.

As discussed in Sect. 5, the main complication is the semantics of (object) invariants. JML's visible state semantics, which ESC/Java2 tries to check (ESC/Java2 is not guaranteed to be sound in this respect), is often stronger than we really need or want. The need for more flexible ways for dealing with invariants is of course widely recognised. Spec# [1] provides one such an approach, and alternatives are systematically compared in [6].

It is very important that the API specifications used are not too rich (i.e. too expressive) and *only* specify the aspects relevant for the navigation graphs. Otherwise the job of annotating and verifying the midlet can become *much* more complicated. Also, as discussed in Sect. 5, Singleton classes occur often in the MIDP API and in typical midlets. Specifying this in JML is a bit clumsy, and (hence) verification with ESC/Java2 seems more complicated than it needs to be. Better ways of dealing with this (possibly by additional primitives in JML) could simplify matters substantially.

A major caveat in our work is that we ignore concurrency. However, the concurrency patterns used in midlets are very simple – essentially, implementations of `commandAction` sometimes start up a worker thread to hand back control to the GUI as soon as possible – so might well be verifiable using a simple approach.

Midlet navigation graphs express safety properties of code, constraining the possible behaviour. This means that crashing of a midlet, say with a `NullPointerException`, can never violate the policy expressed by a navigation graph. This might allow verification to be simplified further: if we can guarantee that code never catches say `NullPointerExceptions` – which could be checked using a simple static analysis – then in the verification we could safely ignore the possibility of `NullPointerExceptions`. This is supported by some verification tools, e.g. the KeY tool [3] offers an option for such simplified reasoning.

PCC. The overall goal of the EU project Mobius, in which this research was carried out, was to provide a Proof-Carrying Code (PCC) framework for Java on mobile devices [2]. PCC [17] involves (i) some security policy, (ii) some untrusted code, and (iii) a proof that this code obeys the policy that can be checked.

To ultimately use our approach in a PCC scenario, it has to be possible to distinguish the JML annotations expressing the desired security policy (i.e. the navigation graph) from any additional JML annotations that are needed for the verification to go through. For the former we must check that these really express the security policy we want. For the latter we do not: we don't care what these are, as long as the proof goes through.

It seems possible to make this distinction here. However, the JML annotations expressing the desired security policy – the navigation graph – do have to refer to program variables (namely the `Displayables` that the program uses), so we cannot quite have these annotations – our “policy” – completely independent of the midlet code.

Also, a malicious midlet could contain specification statements (`set`-statements) that affect the values of the ghost state used in the API specification, making any certification meaningless; it would have to be checked that there are no such statements in the midlet code.

Of course, to then verify and provide PCC certificates for midlets, instead of using ESC/Java2, one should use a sound verification approach that can provide proofs (certificates) [2].

Acknowledgements. This work is supported by the MOBIUS project in the Information Society Technologies programme of the European Commission and the CHARTER project in the ARTEMIS Embedded Computing Systems Initiative. We would also like to thank the anonymous reviewers for their insights.

References

1. Barnett, M., Leino, K., Schulte, W.: The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 151–171. Springer, Heidelberg (2005)
2. Barthe, G., Crégut, P., Grégoire, B., Jensen, T., Pichardie, D.: The MOBIUS proof carrying code infrastructure. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 1–24. Springer, Heidelberg (2008)
3. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
4. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 342–363. Springer, Heidelberg (2006)
5. Crégut, P.: Extracting control from data: User interfaces of MIDP applications. In: Barthe, G., Fournet, C. (eds.) TGC 2007 and FODO 2008. LNCS, vol. 4912, pp. 41–56. Springer, Heidelberg (2008)
6. Drossopoulou, S., Francalanza, A., Müller, P., Summers, A.: A unified framework for verification techniques for object invariants. In: Ryan, M. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 412–437. Springer, Heidelberg (2008)
7. Flanagan, C., Leino, K., Lillibridge, M., Nelson, G., Saxe, J., Stata, R.: Extended static checking for Java. In: PLDI 2002, pp. 234–245. ACM, New York (2002)

8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley, Reading (1999)
9. Hubbers, E., Oostdijk, M.: Generating JML specifications from UML state diagrams. In: Forum on Specification & Design Languages FDL 2003, pp. 263–273. ECSI (2003)
10. Janota, M., Grigore, R., Moskal, M.: Reachability analysis for annotated code. In: SAVCBS, pp. 23–30. ACM, New York (2007)
11. Kiniry, J., Cok, D.: ESC/Java2: Uniting ESC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 108–128. Springer, Heidelberg (2005)
12. Kiniry, J., Morkan, A.E., Denby, B.: Soundness and completeness warnings in ESC/Java2. In: SAVCBS 2006, pp. 19–24. ACM, New York (2006)
13. Leavens, G., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P.: JML reference manual (2003-2007), <http://www.jmlspecs.org>
14. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A Notation for Detailed Design. Kluwer Academic Publishers, Dordrecht (1999)
15. Mobius. Deliverable D5.1 – Selection of case studies. Mobius (2005), <http://mobius.inria.fr>
16. Möller, M., Olderog, E., Rasch, H., Wehrheim, H.: Linking CSP-OZ with UML and Java: A case study. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 267–286. Springer, Heidelberg (2004)
17. Necula, G.C.: Proof-carrying code. In: POPL, pp. 106–119. ACM, New York (1997)
18. Pichardie, D.: Bicolano: a Java bytecode semantics in Coq. (2006), <http://mobius.inria.fr/twiki/bin/view/Bicolano>
19. Pierik, C., Clarke, D., de Boer, F.S.: Creational invariants. In: ECOOP Workshop on Formal Techniques for Java-like Programs, FTfJP 2004 (2004)
20. The Java Verified Program. Unified Testing Criteria for Java technology-based applications for mobile devices, version 3.0 (2009)
21. Trentelman, K., Huisman, M.: Extending JML specifications with temporal logic. In: Kirchner, H., Ringeissen, C. (eds.) AMAST 2002. LNCS, vol. 2422, pp. 334–348. Springer, Heidelberg (2002)

Runtime Verification for Generic Classes with CONGU2

Pedro Crispim, Antónia Lopes, and Vasco T. Vasconcelos

LaSIGE and Faculty of Sciences, University of Lisbon,
Campo Grande, 1749-016 Lisboa, Portugal
{pedro.crispim,mal,vv}@di.fc.ul.pt

Abstract. Even though generics became quite popular in mainstream object-oriented (OO) languages, approaches for checking at runtime the conformance of such programs against formal specifications still lack appropriate support. In order to overcome this limitation within CONGU, a tool-based approach we have been developing to support runtime conformance checking of Java programs against algebraic specifications, we recently proposed a notion of refinement mapping that allows to define correspondences between parametric specifications and generic classes. Based on such mappings, we also put forward a notion of conformance between the two concepts. In this paper we present how the new notion of conformance is supported by version 2 of the CONGU tool.

1 Introduction

The formal specification of software components is an important activity in the process of software development, insofar as specifications are useful, on the one hand, to understand and reuse software and, on the other, to automatically verify the correctness of components implementations. Among the several approaches that can be adopted for automatically analysing the reliability of software components one finds *runtime verification*. This approach involves the monitoring and analysis of system executions. As the system executes, the behaviour of its components is tested for correction with respect to the specification. Runtime monitoring has the advantage that can be used to analyse properties for which static verification fails. Moreover, it does not require the user expertise and effort typically required of a static verification system.

Although generics became quite popular in mainstream OO languages, existent approaches for runtime checking the conformance of generic OO programs against formal specifications still lack appropriate support. This was also the case of CONGU, a tool-based approach to runtime verification of Java implementations against algebraic specifications [10,17]. CONGU is intensively used by our undergraduate students in the context of a course on algorithms and data structures for checking abstract data types (ADTs) implementations.

Given that generics became extremely useful and popular in the implementation of ADTs in Java, in particular those that are traditionally covered in such courses, the lack of support for generics became a major drawback. In order to overcome this limitation, we recently proposed a notion of refinement mapping that allows to define correspondences between parameterized specifications and generic classes [16]. Based on such mappings, we also put forward a more comprehensive notion of conformance between

Java programs and algebraic specifications. This work paved the way for the extension of runtime conformance checking to a more comprehensive range of situations. In this paper, we present a new approach to runtime conformance checking of Java implementations against specifications (applicable to parameterized specifications) and discuss how this solution is realized in the new version of CONGU tool.

The solution for runtime checking that was developed in order to accommodate generics is substantially different from that used before in CONGU [17]. Therein, the strategy was to replace the original classes by proxy classes and generate further classes annotated with monitorable contracts, written in JML [14]. The main innovative aspects of the solution adopted in CONGU2 are the following:

- Introduction of new mechanisms that allow to deal with generics, namely to check whether classes used to instantiate the parameters of generic classes conform to what was specified in the parameter specifications.
- Original classes are not replaced by generated proxy classes. Instead, the solution now relies on the instrumentation of the bytecode of original classes, overcoming the difficulties on the generation of appropriate proxy classes for classes making use of, e.g., public fields or inner classes.
- JML, which does not support generics (among other features introduced in Java 1.5 [9]), is no longer used. Instead, runtime checking of the specified properties at specific execution points is now achieved directly by the generated code, relying only on Java assertions. The compilation with *jmlc* of contract annotated classes was a bottleneck in terms of performance and, with the new solution, we were able to substantially reduce the compile time.

The remainder of the paper is organised as follows. In Section 2 we provide an overview of the CONGU approach, namely we introduce specifications and refinement mappings adopted in CONGU. Then, Section 3 presents the notion of conformance between specifications and Java classes and discusses the properties induced by specifications that are monitored at runtime. The solution for the monitoring of these properties that is realized in CONGU2 is presented in Section 4. Section 5 concludes the paper.

2 Overview of the CONGU Approach

As mentioned before, CONGU supports the runtime conformance checking of Java programs against algebraic specifications. In this section we provide an overview of the CONGU approach, focusing on some of the aspects that are visible to users: the specification language and the notions of specification modules and refinement mapping (see [16] for details). This is achieved by means of an example around a simple ADT — *lists with merge*.

This ADT represents lists composed of “mergeable” elements and that have an operation — *mergeInRange* — that merges the elements of the list in a given range $i \dots j$ (the resulting element is placed in the position i of the list). Figure 1 shows the three elements involved in the specification of this ADT using CONGU’s specification language. In most aspects the language closely follows CASL [4], which is considered a standard for algebraic specification.

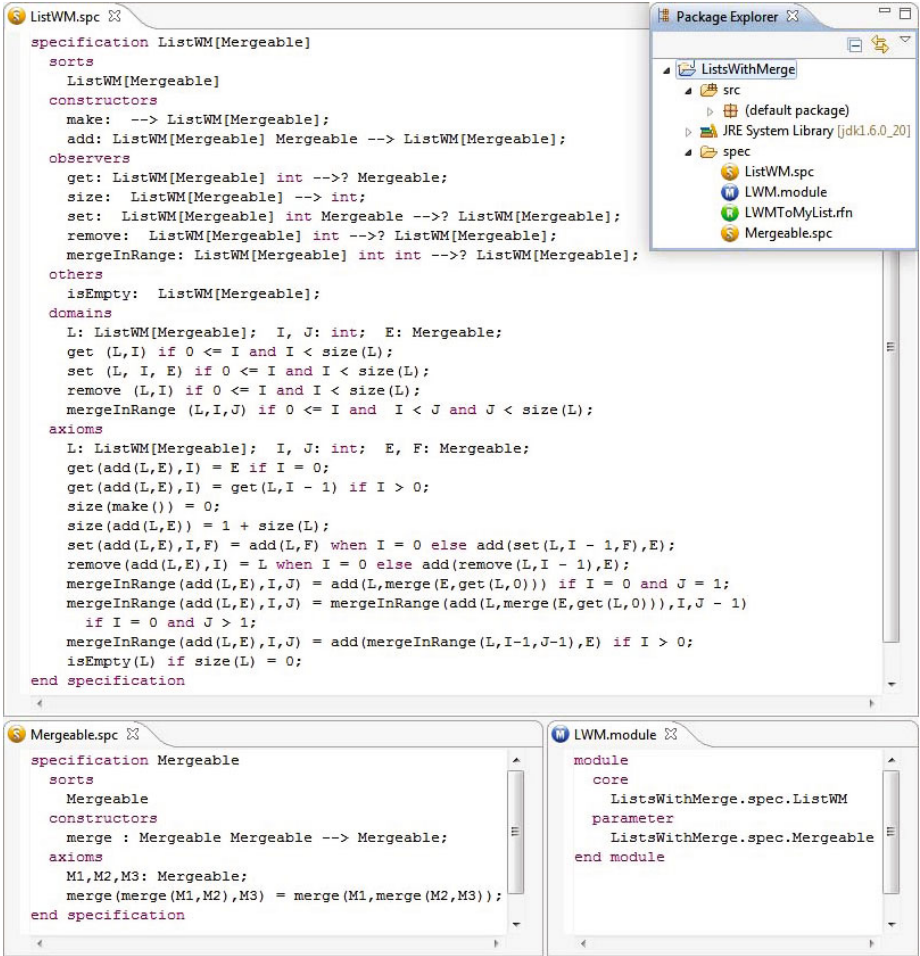


Fig. 1. The three elements involved in the specification of *lists with merge* ADT

The specification ListWM presented in Figure 1 is an example of a parameterized specification. Its parameter is the specification Mergeable, also presented in the figure. Each specification introduces a sort. In our example, Mergeable introduces a simple sort named after it while ListWM introduces the parameterized sort ListMW[Mergeable]. The sort `int`, representing the domain of integer numbers, is primitive in the language.

Then, each specification declares three sets of operations and predicates. Operations declared as constructors are those from which all values of the introduced sort can be built. The other two sets include the operations and also predicates that provide fundamental information about the values of the sort, or are redundant, but useful, operations. The sort of the first argument of these operations is required to be the introduced sort. The difference between the two groups is only on the syntactical structure of the axioms that can be used to define their properties. Axioms for observers are required to be expressed in terms of their application to constructors as first argument and variables

as the remaining arguments; axioms for others are less restrictive, allowing them to be expressed in terms of their application to constructors or variables as first argument and without restriction for the remaining arguments.

Because operations can be partial, specifications also define the *domain condition* of every partial operation, i.e., the situations in which the operation is required to be defined. For instance, in ListWM, `get` is declared to be partial (as indicated by the partial arrow $-->?$) and its domain condition defines that `get(L, l)` must be defined if `l` is indeed an index of list `L`.

As shown in Figure 1, specifications are put together using *specifications modules*. Specifications identified as *core* define the data types that need to be implemented while the role of *parameter* specifications is simply to impose constraints over their admissible instantiations. Now, suppose we have a candidate implementation for module LWM and that we would like to check its conformance against what was specified. First we need to establish a correspondence between each core sort s of the module LWM and a Java type T defined by one of our classes. Moreover, we need to establish a correspondence between the operations and predicates of the specification that introduces s and the methods and constructors of T . In CONGU, this correspondence is defined by means of a *refinement mapping*. In order to capture the role of parameter specifications, these mappings also allow to link parameter specifications with the type variables of generic classes. More concretely, a refinement mapping also defines a correspondence between each parameter sort s and a Java type variable E and also a method signature for each operation/predicate of the specification that introduces s .

Suppose that our candidate implementation for LWM consists of the generic class `MyListWMerge` and the generic interface `Mergeable` presented in Figure 2. The correspondence between LWM and this candidate implementation is defined in the refinement mapping presented in Figure 3. It maps the compound sort `ListWM[Mergeable]` (Figure 1) into the generic type `MyListWMerge<E extends Mergeable<E>>` (Figure 2) and the operations and predicates of the former into methods and constructors of the latter. For instance, we can see that operation `add` is mapped into the method `void addFirst(E e)`. The first argument of an operation always correspond to object `this` and, hence, an operation with n arguments is mapped into a method with arity $n - 1$. Only operations declared constructors whose first argument is not the sort being specified can be mapped into class constructors. Predicates are necessarily mapped into boolean methods. For operations that produce elements of the sort being specified, the corresponding method can either be `void` or of the corresponding type. In this way, it is possible to deal with different implementation styles, namely immutable and mutable implementations. In our example, the class `MyListWMerge` provides a mutable implementation of lists and, hence, all operations in this situation are mapped to `void` methods.

The mapping in Figure 3 also establishes a correspondence between sort `Mergeable` and the type variable `E`. It is defined that the operation `merge` corresponds to the method signature `E merge(E e)`. As we will explain in the next section, this refinement mapping is only correct if the instantiation of `E` in `MyListWMerge<E>` is limited to classes `C` that have a method with signature `C merge(C e)` (which is indeed the case because `E` has `Mergeable<E>` as an upper bound).

```

Mergeable.java
public interface Mergeable<E> {
    E merge (E e);
}

MyListWMerge.java
import java.util.ArrayList;

public class MyListWMerge<E extends Mergeable<E>>{
    private ArrayList<E> list;

    public MyListWMerge() {[]
    public void addFirst(E e) {
        list.add(0,e);
    }
    public boolean contains(E e) {
        return list.contains(e);
    }
    public void mergeInRange(int i, int j) {
        if (i<j){
            E element = list.get(i);
            for(int k=i+1; k<j; k++)
                element = element.merge(list.get(k));
            list.set(i, element);
            for(int k=i+1; k<j; k++)
                list.remove(k);
        }
    }
}

```

Fig. 2. The interface `Mergeable<E>` and an excerpt of the Java class `MyListWMerge<E>`

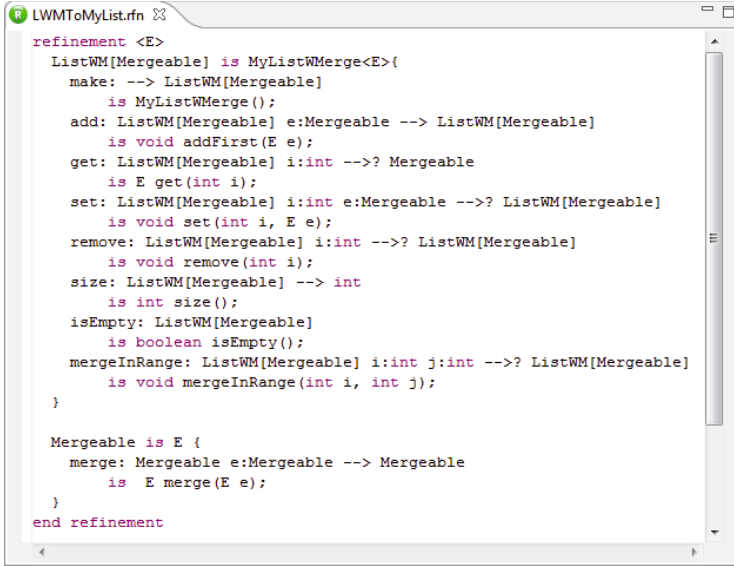
After defining the refinement mapping from module LWM to our candidate implementation, CONGU instruments `MyListWMerge.class` so that, during the execution of any program that uses `MyListWMerge`, the behaviour of this class (made precise in the next section) is checked against what was specified in `ListWM[Mergeable]`. Moreover, the behaviour of the classes used for instantiating `E` in the creation of objects of `MyListWMerge<E>` is also checked against what was specified in `Mergeable`. Suppose, for instance, that we have a program that includes a class `Color` that implements `Mergeable<Color>` and, another class, that creates and manipulates objects of type `MyListWMerge<Color>`. In this case, the behaviour of `Color` is checked against what was specified in `Mergeable`.

3 Runtime Conformance of Programs against Modules

In this section, we present the notion of conformance of Java programs against specification modules that is considered in CONGU2 and discuss some key aspects of CONGU approach to the runtime checking of this notion of conformance.

3.1 Object Properties Induced by Specifications

The conformance of a Java program against a specification module can only be defined if a direct connection between the specifications of the module and the classes of the Java program is provided. As discussed before, in CONGU, the correspondence between



```

LWMToMyList.rfn
refinement <E>
  ListWM[Mergeable] is MyListWMerge<E>{
    make: --> ListWM[Mergeable]
    is MyListWMerge();
    add: ListWM[Mergeable] e:Mergeable --> ListWM[Mergeable]
    is void addFirst(E e);
    get: ListWM[Mergeable] i:int -->? Mergeable
    is E get(int i);
    set: ListWM[Mergeable] i:int e:Mergeable -->? ListWM[Mergeable]
    is void set(int i, E e);
    remove: ListWM[Mergeable] i:int -->? ListWM[Mergeable]
    is void remove(int i);
    size: ListWM[Mergeable] --> int
    is int size();
    isEmpty: ListWM[Mergeable]
    is boolean isEmpty();
    mergeInRange: ListWM[Mergeable] i:int j:int -->? ListWM[Mergeable]
    is void mergeInRange(int i, int j);
  }

  Mergeable is E {
    merge: Mergeable e:Mergeable --> Mergeable
    is E merge(E e);
  }
end refinement

```

Fig. 3. A refinement mapping from LWM to $\{\text{MyListWMerge}\langle E \rangle, \text{Mergeable}\langle E \rangle\}$

specifications and classes is established through the use of refinement mappings. In the previous section we have already mentioned some conditions required by refinement mappings, namely those concerning the matching of method signatures with operations and predicates. The complete set of conditions that a mapping has to meet in order to define a refinement mapping is defined below.

A *refinement mapping* consists of a set V (of type variables) equipped with a pre-order $<$ and a refinement function \mathcal{R} that maps:

1. each core simple specification to a non-generic type defined by a Java class;
2. each core parameterized specification to a generic class, with the same arity;
3. each core specification that defines a sort $s < s'$, to a subtype of $\mathcal{R}(S')$, where S' is the specification defining s' ;
4. each parameter specification to a type variable in V ;
5. each operation of a core specification to a method of the corresponding Java type with a matching signature;
6. each operation of a parameter specification to a matching method signature.

Additionally:

7. if a parameter specification S' defines a subsort of the sort defined in another parameter specification S , then it must be the case that $\mathcal{R}(S') < \mathcal{R}(S)$ holds;
8. if S is a parameterized specification with parameter S' , it must be possible to ensure that any type C that can be used to instantiate the parameter of the generic type $\mathcal{R}(S)$ possesses all methods defined by \mathcal{R} for type variable $\mathcal{R}(S')$ after replacing all instances of the type variable $\mathcal{R}(S')$ by C .

Let us consider again the refinement mapping presented in Figure 3. In this case, the set V is the singleton set $\{E\}$ and only the satisfaction of condition 8 requires some reasoning. According to the definition of the class `MyListWMerge`, type variable `E` must extend `Mergeable<E>`, which, in turn, declares method `E merge(E e)`. Hence, the instantiation of `E` is limited to classes `C` that implement `Mergeable<C>` and, hence, it is ensured that `C` possesses a method with signature `C merge(C e)`.

In the sequel, we assume a fixed refinement mapping \mathcal{R} between a specification module \mathcal{M} and a Java program \mathcal{J} . Intuitively, \mathcal{J} is in conformity with \mathcal{M} iff:

- (i) the properties specified in \mathcal{M} and
- (ii) the algebraic properties of the notion of equality

hold in every possible execution of the program \mathcal{J} .

More concretely, the properties of a core specification S impose constraints on the behavior of every object of type $T_S = \mathcal{R}(S)$, whereas the properties of a parameter specification S used in a parameterized specification, say $S'[S]$, impose constraints on the behavior of every object of a type T_S in \mathcal{J} that is used to instantiate the respective type variable of the generic type $\mathcal{R}(S')$. Axioms and domain conditions impose different type of constraints:

Axioms. Every axiom in a specification S defines, for the objects of type T_S , a property that must hold in all client visible-states.

Let us consider, for instance, the first axiom for `get` in `ListWM[Mergeable]`. Let `lwm` be an object of type `MyListWMerge<C>`. This axiom defines that for every non-null expression `e` of type `C`, after the execution of `lwm.addFirst(e)`, the expression `lwm.get(0).equals(e)` evaluates to `true`¹.

Domains. Every domain condition ϕ of an operation op of a specification S defines that, for every object of type T_S , whenever ϕ holds, the invocation of $\mathcal{R}(op)$ must return normally (i.e., does not throw an exception).

The constraints induced by axioms and domain conditions just presented define a notion of conformance. CONGU, by default, uses a stronger notion that, in addition, also imposes restrictions on the clients of the classes T_S , namely when they invoke a method $\mathcal{R}(op)$: it is required that the `null` value is not passed as argument and the domain condition ϕ holds at the time the method $\mathcal{R}(op)$ is invoked.

This stronger notion of conformance is useful for checking that client code does not call methods in situations where it is not possible assess the normal (non-exceptional) return of a method called outside its domain condition. This is however only appropriate in the absence of additional information about the safe calling conditions for such methods. For instance, if the documentation of class `MyListWMerge<E>` says that `void set(E e, int i)` has no pre-condition and simply produces no effect on the state of the list when `i > size()`, the fact that a class in our program calls this method with an argument that violates this condition should not be identified as a problem of conformance between the program and the specification module LWM. For this reason, we found useful to support the two notions of conformance in CONGU2.

¹ We currently assume that the interpretation of sorts does not include the `null` value but, we envisage that, in the future, refinement mappings may define whether this is appropriate or not.

3.2 Checking Object Properties

In CONGU, the strategy for runtime checking the conformance of a program against a specification module consists in checking the properties induced by the axioms at the end of specific methods, determined by the structure of the axioms. Let us first consider the axioms with a left-hand side expressed in terms of the application of operations or predicates to constructors as first argument and variables as the remaining arguments. In this case, the property induced by the axiom is checked at the end of the method that refines the referred constructor. All method invocations that are performed in order to check a property make use of clones, whenever cloning is possible. Otherwise, the side effects of these methods would affect the monitored objects (if method `clone()` is not available, it is assumed that the class's objects are immutable). For instance, the property induced by the first axiom for `get` is checked at the end of `void addFirst(E e)` through the execution of the following code, where `eOld` is a copy of `e` obtained at the entry of the method.

```

if (eOld != null) {
    E e2 = this.clone().get(0);
    assert(e2 != null && e2.equals(eOld));
}

```

Similarly, the second axiom of the `get` operation is checked by the below code, where `rangeOfInt`, of type `Collection<Integer>`, is populated with integers that cross the boundary (either as parameters or as returned values) of some method in class `MyListWMerge`.

```

if (eOld != null)
  for (int i: rangeOfInt)
    if (i>0 && i<this.clone().size()) {
      E e2 = this.clone().get(i);
      assert((i-1)>=0 && (i-1)<thisOld.clone().size());
      E e3 = thisOld.clone().get(i-1);
      assert(e2!=null && e3!=null && e2.equals(e3));
    }
}

```

On the other hand, the properties induced by axioms that feature a variable as first argument are checked at the end of the method that refines the corresponding operation/predicate. For instance, the last axiom for `isEmpty` is checked at the end of method `boolean isEmpty()` by:

```

if (result) assert(thisOld.clone().size()==0);

```

where `result` is the return value of method `isEmpty()`.

Equality of integers and booleans is translated into comparisons with `==` whereas the equality of terms of a non-primitive sort, say s , is translated into invocation of method `equals` of the class T_S . Therefore, it is essential that all involved classes define a proper implementation of `equals`. Namely, because equality of terms is a congruence, `equals` should be defined in such a way objects are considered equal only if they are behaviourally equivalent with respect to the methods that refine some operation of s (i.e., calling these methods over equal objects must produce equal results).

Correctness of `equals` is checked at runtime as follows. At the end of the method, if the return value is true, then it is checked that by applying the method that refines each observer to the two objects, we obtain equal results. For instance, checking the

correctness of `boolean equals(Object other)` in `MyListWMerge` includes the below code for the `get` operation.

```

if (result)
  for (int i: rangeOfInt)
    if (i>=0 && i<thisOld.clone().size()
        && i<otherOld.clone().size()) {
      E e1 = thisOld.clone().get(i);
      E e2 = otherOld.clone().get(i);
      assert(e1!=null && e2!=null && e1.equals(e2));
    }

```

In addition, at the end of method `clone()`, it is checked that the returned object is equal to the original. Additional properties such as symmetry and transitivity of `equals()` can also be checked, in a similar way, at this point.

Finally, checking that client classes are well-behaved, i.e., do not invoke a method that refines an operation when its domain condition does not hold and also do not pass `null` as argument, can be easily performed at the beginning of the method. For instance, in the case of method `void set(int i, E e)`, this is checked by:

```

assert(e != null && i >= 0 && i < this.clone().size());

```

4 The New CONGU Tool

The new CONGU tool, which we named CONGU2, implements the runtime checking approach conformance of Java classes against specifications described in the previous section. As shown in Figure 4, the tool takes as input a specification module, a set of specifications, a refinement mapping and a Java program (in bytecode form). The program is then transformed so that, when executed under CONGU, the behavior of each class is checked against the corresponding specification. This is achieved by intercepting the calls to all methods that, according to the refinement mapping, refine some operation, and dispatching them to property-monitoring classes generated by the tool.

As mentioned in the introduction, in the previous version of CONGU, the interception of methods was accomplished by proxy classes that wrapped and mimicked the original class's interface as close as possible, capturing the client method calls and diverting them to classes annotated with JML contracts. These contracts were checked at runtime with resort to JML's Runtime Assertion Checker. In order to overcome the difficulties on the generation of appropriate proxy classes for classes making use of more advanced features of the Java language, such as public fields or inner classes, CONGU2 intercepts client method calls through bytecode instrumentation. On the other hand, JML is no longer used and, instead, properties to be checked are now encoded using Java assertions. In this way, CONGU was released from several limitations imposed by the use of JML, namely the lack of support for features introduced in Java 1.5 (generics included), the long compilation times imposed by *jmlc* (namely, because of the compilation of the JML models of the Java API) and the poor and unstable information about assertion violations that hindered the connection of errors with the axioms of specifications (the CONGU solution for this problem, developed for JML 5.4 quickly became obsolete).

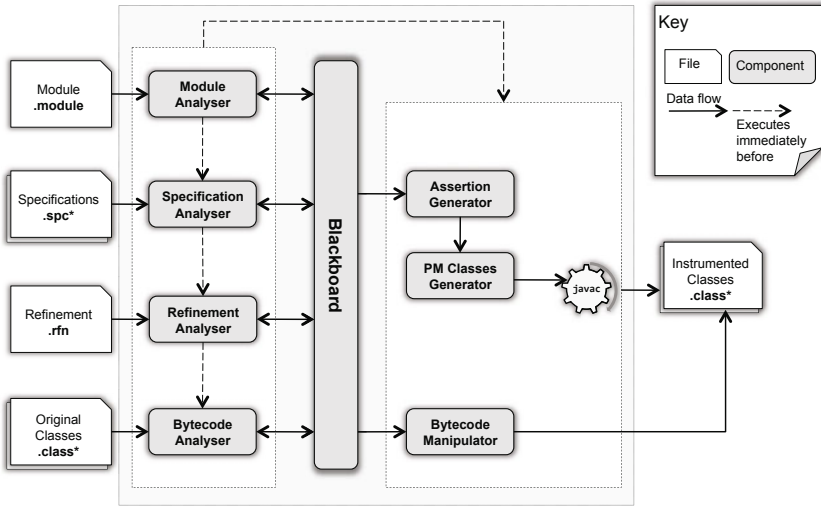


Fig. 4. Overview of CONGU2 architecture

As shown in Figure 4 there are two main tasks in the CONGU support of runtime checking of Java programs against specifications: the analysis of the different input sources and the synthesis of output classes. In the analysis task, the major challenges posed by the extension of the approach to parameterized specifications and generic classes arise in the analysis of refinement mappings.

4.1 Analysis of Refinement Mappings

The extension of the CONGU approach to specification modules including parameterized specifications introduces various challenges in what concerns the analysis of refinement mappings. As detailed in Section 3, refinement mappings impose restrictions on the Java types to which they refer. Enforcement of these restrictions requires querying the Java binaries of the respective classes. This is achieved by taking advantage of Java's reflection facilities, provided by the `java.lang.reflection` package of the Java API.

The process of analyzing refinement mappings comprises two phases. The first phase focuses on the Java classes that refine core specifications while the second addresses the verification of the conditions related with parameter specifications.

Verifying that a non-generic type has the methods mentioned in the refinement is straightforward: it involves querying for the specified method and then checking whether the return type is the expected one. Matters complicate when generic types are at play. Generics in Java are mostly a source code artefact. Simply put, the compiler erases the generic type information, a mechanism known as *type erasure*. Thus, a once generic type becomes a simple type, i.e., a raw type, where all uses of its generic type variables are replaced by their respective upper bounds [11]. For this reason, built-in support for querying classes for methods is limited to signatures defined only in terms of raw

types and, hence, a new strategy for verifying that generic classes possess the methods mentioned in the refinement is needed.

Although information about generics is not used at runtime by the JVM, this information is still retained in the bytecode, in the form of metadata and can be queried through Java's reflection API. The strategy to verify that a generic class has the methods mentioned in the refinement mapping involves retrieving its methods, getting the generic parameter types and generic return type of each one and then comparing with what was expected (a recursive process on the structure of the types).

The second phase of the refinement analysis addresses the verification of the conditions concerning the parameter specifications. Recall that, in this case, specifications are not refined into concrete Java types and what is necessary ensuring is that the classes that can be used to instantiate the corresponding type variable have the right methods. For instance, in our example, in this phase it is verified that every class C that can be used to instantiate E in $\text{MyListWMerge}\langle E \rangle$ possesses a method with signature $C.\text{merge}(C\ e)$. This is achieved by going through the upper bounds of E in MyListWMerge (in our case there is a single upper bound but in general types may have more than one). Methods whose signature depends somehow on type E (in our case, $E.\text{merge}(E\ e)$) need only to be searched on the upper bounds types that are themselves generic and dependent of E (in our case, $\text{Mergeable}\langle E \rangle$) while the remainder methods are searched on all upper bounds types. Searching for these methods in generic types follows the strategy described before.

Additionally, the analysis of the refinement mapping also involves ensuring that hierarchy relationships established for specification sorts are maintained when refining to Java types. This is directly accomplished again making use of Java reflection API, which enables us to query a type for its super class and/or implemented interfaces.

4.2 Bytecode Instrumentation

For monitoring the behaviour of Java programs, CONGU relies on the interception of method calls by client classes. In CONGU2, this is achieved with method call interception through bytecode instrumentation of a copy of the original class (the original bytecode remains unchanged so that the program can also be executed normally). The objective of this instrumentation is to inject bytecode instructions in the methods to forward the call to a corresponding method in a property-monitoring class. From the start, the goal was to minimise the impact on the original bytecode, avoiding any undesirable side effects of its faulty manipulation as much as possible, preferring to generate Java code and rely on the compiler for type safety. Realising this new strategy posed interesting challenges, namely:

Inner calls. How to avoid interception of intra-class method calls? Intra-class calls can not be monitored otherwise we obtain a non-terminating program.

Calls from within superclasses. How to prevent interception of calls from within a superclass? Such calls cannot be monitored for the reason above.

Constructors. How to intercept calls to object constructors and redirect them?

Clone and equals. What to do when these methods are not overridden in the class? (Recall that CONGU relies on these methods and there are properties that have to be monitored when they are invoked.)

Answering these questions was central in overcoming the limitations of earlier version of the CONGU tool. The chosen strategy consists in renaming to *m_Original* each method *m* that refines some operation (each method whose external calls we wish to intercept), and placing in its lieu a method *m* with the exact same interface but dispatching the call to a corresponding method in a property-monitoring class. Moreover, all calls to *m* from within the class and inner classes are replaced by calls to *m_Original* and all calls to *m* in each superclasses are replaced also to calls by calls to *m_Original* that are also added to the superclasses. Although constructors are not methods, they can be for the most part treated as such. Hence, the approach towards the interception of constructor calls is identical to that employed for methods, with the safeguard that constructors require initialisation calls, which are removed from the renamed method and inserted in the replacement method.

More concretely, if *m* is a method of a class *C* that refines some operation, then the bytecode instrumentation process involves the following steps:

1. Within *C*, rename method *m* to *m_Original*;
2. Still within *C*, replace invocations to *m* by invocations to *m_Original*;
3. In *C*'s superclasses, replace invocations to *m* by invocations to *m_Original* and generate a method *m*, with the signature of the original, which just forwards the call to *m_Original*;
4. Generate a replacement for method *m*, with signature of the original, that calls the respective method in the property-monitoring class:

```
congu.properties.CPMonitoring.m(this, ...);
```

5. Rename and generate a replacement for method `equals`; if `equals` is not overridden in *C*, first create a such method that delegates into the superclass;
6. Rename and generate a replacement for method `clone`; if *C* does not implement interface `Cloneable` or does not override method `clone` by making it public, a `clone_Original` method is generated that simply returns `this`. This method is for the exclusive use of the monitoring process.

Implementation of this bytecode-instrumentation approach resorts to the ASM Java bytecode engineering library [6]. ASM is a lightweight and efficient, offering a very simple, well-documented API, full support for Java 6 and an interesting open-source license which allows for convenient packaging within the CONGU2 tool itself.

4.3 Generation of Property-Monitoring Classes

The checking of object properties described in subsection 3.2 is performed in classes generated by the tool, which we call *property-monitoring* classes (or PM-classes, for short). For each Java type *C* under monitoring, there is a corresponding PM-class, named by appending the suffix `PMonitoring` to the name of *C*. In the instrumented bytecode, the intercepted client method calls are dispatched to methods in the respective PM-class.

Each method under monitoring has a counterpart in its respective PM-class, in the form of a static method with the same name, the same return type, the same argument

types. In addition it features an argument `callee` of type C (a reference to original method callee) and a boolean argument `monitoring` (signalling whether monitoring should be performed or not). When invoked from the instrumented bytecode, this flag is set to **true**; when the invocation is realised in the context of a property monitoring it is set to **false**.

These static methods are responsible for the monitoring of the relevant object properties following a general pattern:

1. At the entry of the method, store all elements that, later, are needed for checking some property (these elements are stored in variables starting with *old*).
2. Verify that the client is well-behaved, namely that the domain condition holds (only applicable in the case of strong conformance) and the values passed as arguments are non **null**;
3. Call the original method upon `callee` and keep its return value;
4. Check the properties defined by the axioms.
5. Return the original method return value.

The execution of steps 2 and 4 for method m (which are only executed if flag `monitoring` is true) rely on two separate static methods: `mPre` and `mPos`. These test *callee* for the object properties induced by the specification as described in [3.2](#) but, instead of calling the methods of the original class, they call the method of the corresponding PM-class with `monitoring` set to false.

More concretely, the method `mPre` receives the same arguments as the original one, plus a boolean flag signalling whether to break on a violation, and returning a boolean value, corresponding to whether or not the respective domain condition is satisfied. Method `mPos` is void and takes as arguments: (i) two references to the callee (one before application of the original method, i.e., its *old* value, and another after the original method call); (ii) the arguments of the respective method in the PM-class and (iii) a boolean flag signalling whether to break on a violation.

Figure [5](#) presents a sequence diagram that illustrates the flow of execution in the concrete case of a call to method `mergeInRange(int, int)`.

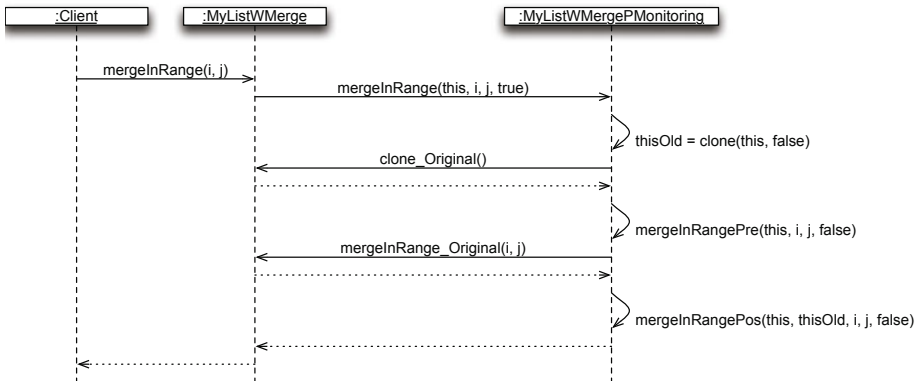


Fig. 5. The property-monitoring process

The monitoring process also heavily depends on an auxiliary method generated as part of each PM-class, named `conguAssert`. This method is responsible for issuing adequate error messages whenever a violation occurs. It takes as parameters the assertion to evaluate, an enumerate value flagging what kind of property was violated (either a domain condition or an axiom-induced property) and error description elements. The method tests the assertion and throws an exception if it is **false**, detailing the violation with the descriptive elements received as arguments. The error description elements passed to this method are the file of the specification to which the property belongs, the domain or axiom to which it pertains as written in the specification and its line. In this way, it is possible to pinpoint the origin of the error, in terms of the specification, which can be invaluable when developing in a specification guided manner.

In step 3, in addition to invocation of the original method, it is also checked that if the domain condition holds, the invocation of the method returns normally. This is achieved by surrounding the original method call with a try-catch statement. The catch clause is only reached when the original method fails to return normally, in which case a violation is issued if the domain condition was true. If the domain condition was false, no constraints apply and, hence, the caught exception is re-thrown, allowing the program to handle it as it would had it been executing normally.

It is worth noting that the properties monitored by each of PM-class do not necessarily originate from a single specification. Refinement mappings do not restrict the number of specifications that a Java type may implement, therefore the PM-class for a given type is responsible for monitoring the properties arising from all the specifications that have been refined to mentioned type.

All of the above holds true for both core and parameter specifications. However, parameter specifications require another level of indirection. Let C be a generic class with a parameter E that refines a core parameterized specification, say $S[S']$. Even though each class that is used to instantiate E in C has a corresponding PM-class, the code generated for monitoring the properties of S that involves to call a method over an object e of type E , cannot commit to a specific PM-class (the actual type of e is only known at runtime and will vary from call to call).

CONGU2's solution is to generate a dispatcher class associated to each type variable of the refinement mapping. This class has the exact same methods of a PM-class, but, instead of monitoring properties, only resolve to which PM-class should the call be forwarded to, based on the actual type of object *callee*. In the PM-class for class C , whenever the testing of a property requires to invoke a method of E , the call is placed to the respective dispatcher class.

Suppose that our program manipulates an object `lc` of type `MyListWMerge<Color>` and another `lt` of type `MyListWMerge<Text>`. Monitoring the behaviour of these objects is performed by the `MyListWMergePMonitoring` class. Whenever such operation involves a call to method `E merge(E)`, we call the respective method in the `EPMonitoring` dispatcher class. While monitoring `lc`'s behaviour, the actual type of the callee is `Color` and, hence, the dispatcher forwards the invocation of `merge` to `ColorPMonitoring`. Similarly, while monitoring `lt`, the calls are forwarded to class `TextPMonitoring`. Notice however that when, as a result of calling `mergeInRange` over `lc`, method `E merge(E)` is called (see the body of the method in Figure 2), this

call is intercepted and forwarded to `ColorPMonitoring` in order for the properties of this operation to be checked.

5 Conclusions

The importance of tools that support runtime conformance checking of implementations against formal specifications has long been recognised. In the last decade, many approaches have been developed for monitoring the correctness of OO programs w.r.t. formal specifications (e.g., [13,7,8,12,15]). However, despite the actual popularity of generics in mainstream OO languages [5], current approaches still lack support such a feature. This was also a limitation CONGU that was overcome in CONGU2.

In this paper we showed how the CONGU approach and the corresponding tool were extended in order to support the specification of parametrized data types and their implementation in terms of generic classes. The extension of the specification language in order to support the description of generic data types was relatively simple. Given that CONGU relies on property-driven specifications, this mainly required the adoption of parameterized specifications available in conventional algebraic specification languages. In order to bridge the gap between parameterized specifications and generic classes we proposed a new notion of refinement mapping around which a new notion of conformance between specifications and OO programs was defined. To the best of our knowledge, this issue has not yet been addressed in other contexts. Other approaches exist that deal with the problem of the implementation of architectural specifications including parameterized specifications as, for example [2], but the target are ML programs. Relationships between algebraic specification and OO programs that we are aware of, namely those that address runtime conformance checking or testing, exclusively consider flat and non-parameterized specifications (e.g., [11,2,18]).

CONGU2 implements the runtime monitoring of this new notion of conformance which, in the case of generic data types, involves checking that both the class that implements the data type and the Java types used to instantiate it conform with what was specified. With CONGU2, runtime conformance checking becomes applicable to a range of situations in which automatic support for detection of errors becomes more relevant. Generics are known to be difficult to grasp and, hence, with generics in action, obtaining correct implementations becomes more challenging. For us, it is particularly important to be able to use CONGU with the generic data types that appear in the context of a typical Algorithms and Data Structures course: we believe this course constitutes an excellent opportunity for exposing undergraduate students to formal methods. As discussed by Hu [13], accurate descriptions of abstract data types, agnostic w.r.t. programming paradigms and languages, are important for teaching these concepts. From our experience in teaching this course for several years (initially without tool support and, more recently, using CONGU), we are convinced that, from an educational and motivational point of view, it is quite important that students experience, in their practice, that they can take real advantage of formal descriptions. The use of a simple tool that allows them to gain confidence that their classes correctly implement a given data type has shown to be a good starting point. The extension of the tool to support generics will contribute to the success and effectiveness of the CONGU approach to the introduction to formal methods in the computer science curriculum.

Acknowledgement

This work was partially supported by FCT through the project QUEST (PTDC/EIA-EIA/103103/2008).

References

1. Antoy, S., Hamlet, R.: Automatically checking an implementation against its formal specification. *IEEE Transactions on Software Engineering* 26(1), 55–69 (2000)
2. Aspinall, D., Sannella, D.: From specifications to code in CASL. In: Kirchner, H., Ringeisen, C. (eds.) *AMAST 2002*. LNCS, vol. 2422, pp. 1–14. Springer, Heidelberg (2002)
3. Barnett, M., Schulte, W.: Runtime verification of .NET contracts. *Journal of Systems and Software* 65(3), 199–208 (2003)
4. Bidoit, M., Mosses, P. (eds.): *CASL User Manual*. LNCS, vol. 2900. Springer, Heidelberg (2004)
5. Bracha, G.: *Generics in the Java programming language* (2004), ava.sun.com/j2se/1.5/pdf/generics-tutorial.pdf
6. Bruneton, E., Lenglet, R., Coupaye, T.: ASM: A code manipulation tool to implement adaptable systems. In: *Proc. ACM SIGOPS France Journées Composants 2002: Systèmes à composants adaptables et extensibles* (2002)
7. Chen, F., Roşu, G.: Java-MOP: A monitoring oriented programming environment for Java. In: Halbwachs, N., Zuck, L.D. (eds.) *TACAS 2005*. LNCS, vol. 3440, pp. 546–550. Springer, Heidelberg (2005)
8. Cheon, Y., Leavens, G.T.: A runtime assertion checker for the Java Modeling Language (JML). In: *Proc. International Conference on Software Engineering Research and Practice (SERP 2002)*, pp. 322–328. CSREA Press (2002)
9. Cok, D.R.: Adapting JML to generic types and Java 1.6. In: *Proc. Specification and Verification of Component-Based Systems Workshop* (2008)
10. *Contract Based System Development*, <http://gloss.di.fc.ul.pt/congu/>
11. Gosling, J., Joy, B., Steele, G., Bracha, G.: *The Java Language Specification*, 3rd edn. Prentice-Hall, Englewood Cliffs (2005)
12. Henkel, J., Diwan, A.: Discovering algebraic specifications from Java classes. In: Cardelli, L. (ed.) *ECOOP 2003*. LNCS, vol. 2743, pp. 431–456. Springer, Heidelberg (2003)
13. Hu, C.: Just say a class defines a data type. *Communications of the ACM* 51(3), 19–21 (2008); see also *Forum in Communications of the ACM* 51(5), 9–10 (2008)
14. Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming* 55(1–3), 185–208 (2005)
15. Meyer, B.: *Object-Oriented Software Construction*, 2nd edn. Prentice-Hall PTR, Englewood Cliffs (1997)
16. Nunes, I., Lopes, A., Vasconcelos, V.T.: Bridging the gap between algebraic specification and object-oriented generic programming. In: Bensalem, S., Peled, D.A. (eds.) *RV 2009*. LNCS, vol. 5779, pp. 115–131. Springer, Heidelberg (2009)
17. Nunes, I., Lopes, A., Vasconcelos, V., Abreu, J., Reis, L.S.: Checking the conformance of Java classes against algebraic specifications. In: Liu, Z., Kleinberg, R.D. (eds.) *ICFEM 2006*. LNCS, vol. 4260, pp. 494–513. Springer, Heidelberg (2006)
18. Yu, B., King, L., Zhu, H., Zhou, B.: Testing Java components based on algebraic specifications. In: *Proc. International Conference on Software Testing, Verification and Validation*, pp. 190–198. IEEE, Los Alamitos (2008)

A High-Level Language for Modeling Algorithms and Their Properties

Sabina Akhtar, Stephan Merz, and Martin Quinson

LORIA – INRIA Nancy Grand Est and Nancy University, Nancy, France
{Sabina.Akhtar,Stephan.Merz,Martin.Quinson}@loria.fr

Abstract. Designers of concurrent and distributed algorithms usually express them using pseudo-code. In contrast, most verification techniques are based on more mathematically-oriented formalisms such as state transition systems. This conceptual gap contributes to hinder the use of formal verification techniques. Leslie Lamport introduced PLUSCAL, a high-level algorithmic language that has the “look and feel” of pseudo-code, but is equipped with a precise semantics and includes a high-level expression language based on set theory. PLUSCAL models can be compiled to TLA⁺ and verified using the model checker TLC.

However, in practice, the use of PLUSCAL requires good knowledge of TLA⁺ and of the translation from PLUSCAL to TLA⁺. In particular, the user needs to annotate the generated TLA⁺ model in order to define the properties to be verified and to introduce fairness hypotheses. Moreover, the PLUSCAL language enforces certain restrictions that often make it difficult to express distributed algorithms in a natural way. We propose a new version of PLUSCAL with the aim of overcoming these limitations, and of providing a language in which algorithms and their properties can be expressed naturally. We have implemented a compiler of our language to TLA⁺, supporting the verification of algorithms by finite-state model checking.

1 Introduction

Algorithms for concurrent and distributed systems [11] are notoriously hard to design, due to the number of interleavings of their constituent processes that must communicate and synchronize properly in order to achieve the desired function. It is all too easy to overlook corner cases, and hard to generate or reproduce particular behaviors during testing. Formal verification of such algorithms is therefore essential, and model checking in particular has been applied with great success in this context. However, there is a conceptual gap between the languages algorithm designers use to convey their ideas and the input languages of model checking tools. While the former emphasize high levels of abstraction in order to present the algorithmic ideas, their semantics is not precisely defined. Languages for model checkers come with a more precise (at least operational) semantics but tend to make compromises in terms of the available data types in order to enable compact state representations and the efficient computation

of operations such as the computation of successor (or predecessor) states. Most model checkers, in particular symbolic ones, support only low-level data types such as fixed-size integers and records. TLC [14], the model checker for the specification language TLA^+ [8], accepts a significant fragment of TLA^+ , which is based on set theory; it thus provides one of the most expressive and high-level input languages for model checking. However, TLA^+ models encode transition systems via logical formulas, losing much of the (control) structure that is present in code.

Recently, Lamport introduced the PLUSCAL algorithm language [9] (originally called +CAL). While retaining the high level of abstraction of TLA^+ expressions, it provides familiar constructs of imperative programming languages for describing algorithms, such as processes, assignments, and control flow. The PLUSCAL compiler generates a TLA^+ model corresponding to the PLUSCAL algorithm, which is then verified using TLC. PLUSCAL is a high-level language that features set-based abstractions, non-determinism, and user-specified grain of atomicity; it emphasizes the analysis, not the efficient execution of algorithms and aims at bridging the gap that we described above.

Unfortunately, as we discuss in more detail in section 2, use of Lamport’s PLUSCAL requires good knowledge of TLA^+ , and even of the translation of PLUSCAL to TLA^+ . Aiming at a simple translation in order to make the resulting TLA^+ model human readable, Lamport imposed some limitations on the language that can make it difficult or unnatural to express distributed algorithms. After initial attempts to extend the original language and its compiler, these limitations motivated us to develop a new version of PLUSCAL that retains the basic ideas of Lamport’s language but overcomes the shortcomings that we identified. At the same time, we aim at a translation that enables the use of reduction techniques and hence more efficient verification.

2 Evaluation of PlusCal

TLA^+ is a very expressive specification language that emphasizes the use of high-level constructs such as sets and functions for expressing algorithms. A TLA^+ module contains a list of declarations, assertions, and definitions. In particular, an algorithm is specified as a formula of temporal logic that describes which executions are permitted. PLUSCAL retains the logical basis and the expression language of TLA^+ , but otherwise resembles a (pseudo) programming language for multi-process programs, extended by non-deterministic constructs useful for modeling algorithms. In our experience we found that thinking in terms of sets is a strong point of PLUSCAL, as it can make the description of algorithms much more perspicuous. However, as we explain now, we also identified a number of shortcomings when trying to use PLUSCAL for modeling distributed algorithms.

Need to understand TLA^+ and the compilation. PLUSCAL models are not fully self-contained: the algorithm is described in the PLUSCAL language, but PLUSCAL can express neither the correctness properties that should be verified nor

fairness assumptions assumed about the algorithm's execution, which underly the proof of liveness properties. Rather, the user must add these as temporal logic formulas to the module generated by the PLUSCAL compiler. It is therefore necessary to understand not only TLA⁺, but also the translation of PLUSCAL to TLA⁺. An effort was made in the design of the PLUSCAL language to keep the translation simple. For example, the compiler tries to preserve the names of PLUSCAL variables in the TLA⁺ specification. However, this is not always possible, for example if variables of the same name are declared in different procedures. Also, local variables of processes are represented as arrays in TLA⁺, and the user must be aware of this when annotating the TLA⁺ model.

Lack of process hierarchy and of scoping. Another serious restriction motivated by the need for a simple translation is that PLUSCAL processes can only be declared at top level, without any nesting. As we will illustrate in section 3 using Lamport's distributed mutual exclusion algorithm, many distributed algorithms are more naturally expressed using hierarchies of processes.

A related issue is the lack of scoping rules in PLUSCAL. Although variables may be declared locally to processes, scoping is not enforced and local variables can in fact be accessed throughout the program. Beyond being a possible cause of errors, the lack of a proper hierarchy of processes and of scoped local variables makes it much more difficult to implement optimizations for verification, such as partial-order reduction.

Restrictions in specifying atomicity. An important concern in modeling concurrent and distributed algorithms is the specification of the proper unit of atomicity: which (blocks of) statements can be considered to be executed without interleaving with statements of other processes? Whereas too coarse-grained atomicity may hide errors that arise in the implementation due to unexpected interleavings, too fine-grained atomicity introduces unnecessary details and causes state space explosion in verification. PLUSCAL uses a simple but powerful idea for expressing atomicity: the user may decorate statements with labels, and interleaving is only allowed at labeled statements. However, the user is not entirely free where labels may or may not be placed, as these also serve for internal purposes of compilation. Typically, more labels must be introduced than would be necessary, hence aggravating state space explosion.

Technical limitations. Lamport's PLUSCAL imposes a number of other limitations, again motivated by the desire to keep the translation simple. For example, although sets are the basic construct for representing data, PLUSCAL does not contain a primitive for iterating over the elements of a set. The programmer has to introduce an auxiliary variable for iteration and keep track of the elements that have already been handled. Without special care, these auxiliary variables will again lead to state space explosion during model checking. Another technical restriction enforced in PLUSCAL is to disallow multiple assignments to the same variable within an atomic step.

3 Introducing a New Version of PlusCal

We now present in some more detail the main features of our version of PLUSCAL and describe its compilation to TLA⁺. From now on, PLUSCAL will denote our version, except if explicitly stated otherwise.

3.1 Structure of an Algorithm

Figures 1 and 2 show a model of Lamport’s mutual exclusion algorithm 6 in PLUSCAL. This is a basic distributed algorithm and shows some of the main ingredients of the language.

The *header section* indicates the name of the algorithm and lists any TLA⁺ modules to be imported (“extended”). These modules contain definitions of operators that are used within the algorithm. Algorithm `LamportMutex` imports the modules `Naturals` and `Sequences` from the TLA⁺ standard library. Global constant parameters (`N` and `maxClock` in our example) are also declared in the header section; these will later be instantiated to obtain a finite-state instance for verification.

The following *declaration section* contains declarations of global variables, procedures, and definitions. As in TLA⁺, variables are untyped. A variable declaration may provide an initial value. In our example, we declare a global variable `network` as a two-dimensional array indexed by elements of the site `Site`, which contains the identities of the processes of type `Site`, declared below. The variable `network` represents the communication network; more precisely, `network[from][to]` is a queue of messages sent from site `from` to site `to`, initially empty ($\langle \rangle$ denotes the empty sequence in TLA⁺). The operators `send` and `broadcast`, defined next, model point-to-point and broadcast communication over the network. Specifically, `send` computes the network obtained by sending a single message between two processes¹, and `broadcast` computes the state of the network after site `from` sends a message to all sites.

The main part of a PLUSCAL program consists of the *process section*, which introduces the processes that participate in the algorithm. Programs can declare any number of process types, and processes can be nested. Since we are mainly interested in finite-state model checking, all process instances are created at initialization time and we do not provide a mechanism for process activation at run time. In the example, we declare that `N` processes of type `Site` will be run, each containing one instance of process `Communicator`. Processes may be declared as being *fair*; for example, we assume (weak) fairness for each instance of process `Communicator`. Each process contains declarations of local variables, procedures or definitions analogously to the global declaration section. These declarations are properly scoped and visible only within the enclosing process. In our example, process `Site` declares variables `clock`, `reqQ`, and `acks` that represent the value of its logical clock, the sequence of requests it has received (which will be ordered by timestamp), and the set of acknowledgements it has received for

¹ The short-hand `@` denotes the current value of the array cell being assigned to.

```

1 algorithm LamportMutex
2 extends Naturals, Sequences      (* standard modules *)
3 constants N, maxClock
4
5 variable network = [from ∈ Site ↦ [to ∈ Site ↦ ⟨⟩]]
6 definition send(from, to, msg)  $\triangleq$ 
7   [network EXCEPT ![from][to] = Append(@, msg)]
8 definition broadcast(from, msg)  $\triangleq$ 
9   [network EXCEPT ![from] = [to ∈ Site ↦ Append(network[from][to], msg)]]
10
11 process Site[N]
12   variables
13     clock = 1,      (* logical clock of this site *)
14     reqQ = ⟨⟩,      (* queue of pending requests, ordered by clock values *)
15     acks = {}      (* set of acknowledgements received for own request *)
16   definition beats(rq1, rq2)  $\triangleq$ 
17      $\vee$  rq1.clk < rq2.clk
18      $\vee$  rq1.clk = rq2.clk  $\wedge$  rq1.site < rq2.site
19   definition insertRequest(from, c)  $\triangleq$ 
20     LET entry  $\triangleq$  [site ↦ from, clk ↦ c]
21     len  $\triangleq$  Len(reqQ)
22     pos  $\triangleq$  CHOOSE i ∈ 1 .. len+1 :
23        $\wedge \forall j \in 1 \dots i-1 : \text{beats}(\text{reqQ}[j], \text{entry})$ 
24        $\wedge i = \text{len}+1 \vee \text{beats}(\text{entry}, \text{reqQ}[i])$ 
25     IN SubSeq(reqQ, 1, pos-1)  $\circ$  (entry)  $\circ$  SubSeq(reqQ, pos, len)
26   definition removeRequest(from)  $\triangleq$ 
27     LET len  $\triangleq$  Len(reqQ)
28     pos  $\triangleq$  CHOOSE i ∈ 1 .. len : reqQ[i].site = from
29     IN SubSeq(reqQ, 1, pos-1)  $\circ$  SubSeq(reqQ, pos+1, len)
30   definition max(x,y)  $\triangleq$  IF x<y THEN y ELSE x
31
32 fair process Communicator[1]
33 begin
34   loop
35     rcv: with from ∈ {s ∈ Site : Len(network[s][super]) > 0},
36       msg = Head(network[from][super])
37     do network[from][super] := Tail(@);
38     if msg.kind = "request"
39     then reqQ := insertRequest(from, msg.clk);
40       clock := max(clock, msg.clk) + 1;
41       network := send(super, from, [kind ↦ "ack"]);
42     elseif msg.kind = "ack"
43     then acks := acks  $\cup$  {from};
44     elseif msg.kind = "free"
45     then reqQ := removeRequest(from);
46     end if;
47   end with;
48 end loop;
49 end process (* Communicator *)

```

Fig. 1. Lamport's mutual-exclusion algorithm in extended PLUSCAL (part 1)

```

1  begin (* process Site *)
2    loop
3    ncrit: skip;
4    try:   network := broadcast(self, [kind ↦ "request", clk ↦ clock]);
5           acks := {};
6    +enter: when Len(reqQ) > 0 ∧ Head(reqQ.proc) = self ∧ acks = Sites;
7    +crit:  skip;
8    +exit:  network := broadcast(self, [kind ↦ "free"]);
9    end loop;
10   end process (* Site *)
11  end algorithm
12
13  invariant  $\forall s, t \in \text{Site} : \text{Site}[s]@\text{crit} \wedge \text{Site}[t]@\text{crit} \Rightarrow s=t$ 
14  invariant
15     $\wedge$  (* each queue holds at most one request per site *)
16     $\forall s \in \text{Site} : \forall i \in 1 \dots \text{Len}(\text{reqQ}[s]) : \forall j \in i+1 \dots \text{Len}(\text{reqQ}[s]) :$ 
17      reqQ[s][j].site  $\neq$  reqQ[s][i].site
18     $\wedge$  (* requests stay in queue until "free" message received *)
19     $\forall s, t \in \text{Site} :$ 
20       $(\exists i \in 1 \dots \text{Len}(\text{network}[s][t]) : \text{network}[s][t][i].\text{kind} = \text{"free"})$ 
21       $\Rightarrow \exists j \in 1 \dots \text{Len}(\text{reqQ}[t]) : \text{reqQ}[t][j].\text{site} = s$ 
22     $\wedge$  (* site is in critical section only if at the head of every request queue *)
23     $\forall s \in \text{Site} : \text{Site}[s]@\text{crit} \Rightarrow \forall t \in \text{Site} : \text{Head}(\text{reqQ}[t]).\text{site} = s$ 
24  temporal  $\forall s \in \text{Site} : \text{Site}[s]@\text{enter} \rightsquigarrow \text{Site}[s]@\text{crit}$ 
25
26  (* Finite instance for model checking *)
27  constants N = 3, maxclock = 5
28  constraint  $\forall s \in \text{Site} : \text{Site}[s].\text{clock} \leq \text{maxClock}$ 

```

Fig. 2. Lamport’s mutual-exclusion algorithm (part 2)

its own request (if any). Elements of the request queue are records with fields `site` and `clock` indicating the requesting site and the timestamp of the request. The definitions `beats`, `insertRequest`, and `removeRequest` formalize the priorities between two requests and the insertion and removal of a request in the priority queue.

The code of a process is given in the *code section* between the keywords **begin** and **end process**. We describe the statements of PLUSCAL in more detail in Section 3.2.

The process section is optionally followed by a code section for the main algorithm, which executes in parallel to the processes. No such code section is required for algorithm `LamportMutex`.

The description of the algorithm itself is followed by the *property* and *instance* sections, which state the properties (invariants and general temporal logic properties) to be verified. For our example, we state two invariants and one temporal (liveness) property. The first invariant expresses mutual exclusion between all sites by asserting that no two sites are simultaneously at label `crit`. The second invariant states further safety properties of the algorithm that give more insight in its functioning. The temporal property asserts that whenever some site `s` is

at the statement labeled **enter**, it will eventually access its critical section.² Local properties of single processes may also be stated after the code section of processes; they are verified for every instance of the corresponding process type.

The instance section of a PLUSCAL program determines the finite instance of the model that will be verified by the model checker TLC. In particular, the user must specify concrete values for all constants that have been declared in the header section. In our example, we instantiate the constant parameters `N` and `maxClock` by 3 and 5. This section may also contain other declarations that are interpreted by TLC. In particular, we define a constraint that bounds the state space for model checking by considering only such states where no clock value exceeds `maxClock`.

3.2 PlusCal Statements

The syntax of PLUSCAL resembles that of a standard imperative programming language, but adds non-deterministic constructs, which are useful for modeling. The expressions of PLUSCAL are just TLA⁺ expressions. By focusing on high-level abstractions such as sets and functions, users are encouraged not to commit to any particular implementation early. We have found that algorithm designers quickly learn how to write TLA⁺ expressions. This section introduces the key statements of PLUSCAL.

Basic statements include assignments and the **skip** statement, which has no effect. Statements can be labeled (cf. the labels `ncrit`, `try` etc. in Fig. 2). Lamport's PLUSCAL introduced the key idea that labels define the atomic unit of execution of a PLUSCAL model: a group of statements appearing between two labels is executed atomically, without interleaving by other processes. For example, reception and processing of a message is modeled as being atomic in process `Communicator` of Fig. 11. However, the compiler sometimes introduces additional labels when translating to TLA⁺. For example, the first statement appearing inside a loop or statements following a procedure call must be labeled. Our compiler adds any required labels, but since every label creates an additional point of interleaving, we have added an **atomic** statement to PLUSCAL, which was not present in Lamport's language.

atomic *B* **end atomic**

The statements *B* in this form are executed (pseudo-)atomically, even if they contain labels. TLC can be used to find deadlocks caused by statements inside an atomic block being non-executable.

Case distinction is expressed by a standard **if** statement. There is also a **when** statement that blocks until the specified condition becomes true. Less conventionally, the statement

either *B*₁ **or** ... **or** *B*_{*n*} **end either**

² The TLA⁺ formula $P \rightsquigarrow Q$ asserts that every state satisfying predicate *P* will eventually be followed by a state satisfying predicate *Q*.

can be used to express non-deterministic choice between n possible branches. In fact, the **if**, **when**, and **either** constructs are just special cases of the primitive form

```

branch
   $P_1$  then  $B_1$ 
   $P_2$  then  $B_2$ 
  ...
   $P_n$  then  $B_n$ 
  else  $B$ 
end branch

```

inspired by Dijkstra's guarded commands [3]. The first n branches consist of a condition P_i and a block of statements B_i , the final **else** branch is optional. The effect of a **branch** statement is to non-deterministically choose some i such that P_i evaluates to true and execute the corresponding block B_i . If no P_i is true then the **else** branch is executed if it is present, otherwise execution blocks (and another process may be executed).

Non-deterministic choice over the elements of a set (rather than a fixed number of alternatives) is expressed by the statement

```

with  $x \in S$  do  $B$  end with

```

which executes the statements B for some element of the set S , and blocks if S is empty. Since the expression S may contain program variables, executing the code of other processes may unblock a **with** statement.

PLUSCAL offers three iteration constructs. Beyond the standard **while** loop

```

while  $P$  do  $B$  end while

```

which is already present in Lamport's PLUSCAL, we added the statement

```

loop  $B$  end loop

```

for expressing infinite loops; this simply abbreviates **while** TRUE **do** B . More importantly, we added a loop of the form

```

for  $x \in S$  do  $B$  end for

```

for iterating over the elements of a set S . (In contrast, the **with** statement mentioned above executes its body for a single, nondeterministically chosen element of S .) The order in which the elements of S are processed is unspecified but fixed. Exploring only one iteration order helps mitigate state space explosion, but the correctness of the algorithm should not depend on any particular order [3].

We extended PLUSCAL by the ability to express fairness assumptions for algorithms inside the language rather than through TLA⁺ formulas that the user must add to the generated model. Fairness assumptions are fundamental for the

³ We plan to add a compile-time option to a future version of the PLUSCAL compiler that will cause the model checker to explore all possible iteration orders.

verification of liveness properties. We have already mentioned (weak) fairness annotations for processes in section 3.1; these ensure that whenever a process instance is continually executable, it will eventually proceed. A **strongly fair** process is guaranteed to make progress if it is repeatedly (though not necessarily continually) executable. PLUSCAL also supports fairness annotations attached to individual statements (i.e., labels) rather than to entire processes. For example, the leading “+” signs decorating the labels enter, crit, and exit indicate weak fairness assumptions for the corresponding statements.

3.3 The PlusCal Compiler

The basic idea of the translation of PLUSCAL to TLA⁺ is to represent each group of statements between two labels, and hence executed atomically, by a TLA⁺ action formula. (An action formula contains unprimed and primed copies of state variables, which represent the values of these variables before and after the state transition.) Control flow is made explicit by adding a variable containing the values of the program counters of all process instances and of the main program (if present). Technically, the PLUSCAL compiler proceeds in three phases, as illustrated in Fig. 3.

Parser. The PLUSCAL parser is generated from a JavaCC grammar. Besides analyzing the algorithm for syntactic errors, the parser also constructs a symbol table, maintaining information about declared identifiers and checking that their use respects the scoping rules. This phase generates an abstract syntax tree (AST) that represents the PLUSCAL algorithm.

Translation to intermediate format. For clarity and modularity, we split the compilation into two phases. The first phase makes the control flow explicit and converts the AST to a list of labeled guarded commands (**branch** statements) whose branches contain only assignments. Each branch ends with an explicit assignment to the program counter of the entity executing the statements. For example, the statement

$$\begin{array}{l} \underline{\Delta}: \text{ while } P \\ \quad \text{ do } B \\ \underline{\mu}: \dots \\ \text{ end while} \\ \underline{\nu}: \dots \end{array}$$

(where there are no labels within the group of statements B) would first generate the guarded command

$$\begin{array}{l} \underline{\Delta}: \text{ branch} \\ \quad P \text{ then } \overline{B}; \text{ pc}[\text{self}] := \mu; \\ \quad \neg P \text{ then } \text{pc}[\text{self}] := \nu; \\ \text{ end branch} \end{array}$$

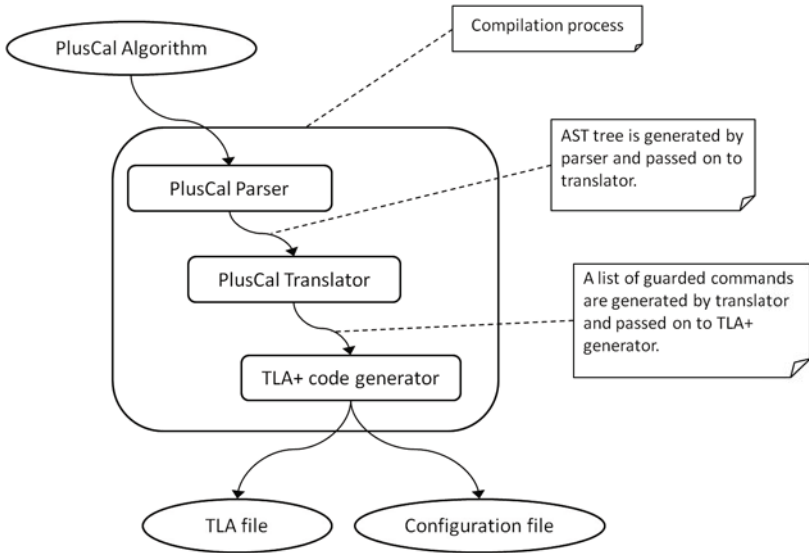


Fig. 3. The compilation phases for the new version of PLUSCAL

where \overline{B} denotes the guarded command corresponding to B . Any nested guarded commands resulting from this translation are subsequently flattened. Procedure calls and returns are handled using an explicit run-time stack.

This translation may require additional labels, in particular for translating loops and returns from procedure calls, and the translator adds those as necessary, and signals labels added in this way to the user.

Generation of TLA⁺ code. The final phase of compilation generates the actual TLA⁺ model from the list of guarded commands obtained from the preceding phase. A guarded command

```

λ: branch
    P1 then x := t; y[self] := u; pc[self] := μ
    ...
    Pn then ...
end branch
    
```

is essentially translated to the TLA⁺ action

$$\begin{aligned}
 \lambda(\text{self}) \triangleq & \wedge pc[\text{self}] = \lambda \\
 & \wedge \vee \wedge P_1 \\
 & \quad \wedge x' = t \\
 & \quad \wedge y' = [y \text{ EXCEPT } ![\text{self}] = u] \\
 & \quad \wedge pc' = [pc \text{ EXCEPT } ![\text{self}] = \mu] \\
 & \quad \wedge \text{UNCHANGED } vars \setminus \{x, y, pc\} \\
 & \vee \dots \\
 & \vee \wedge P_n \\
 & \quad \wedge \dots
 \end{aligned}$$

where *vars* contains all state variables. Multiple assignments to the same variable within a group of statements are handled by introducing intermediate LET-bound constants.

User-defined atomic blocks are implemented using a system-wide lock variable that is acquired at the begin of the block and released at the end. The guards of actions are strengthened appropriately: actions corresponding to statements within an atomic block test whether the lock is held by the current process, the other actions verify that the lock is free. In case some statement within an atomic block may become non-executable during the execution of the algorithm, TLC will signal a deadlock during verification.

After generating all actions corresponding to the individual transitions of the PLUSCAL model we define the transition relation of a process as the disjunction of the actions it may execute (including actions occurring within procedures), and the overall next-state relation as the disjunction of the transition relations for all process instances, and for the main code section if present. For the example of Figs. 1 and 2 we obtain

$$\begin{aligned} Next \triangleq & \vee \exists self \in Site : _Site(self) \\ & \vee \exists self \in Communicator : _Communicator(self) \end{aligned}$$

where *Site* and *Communicator* are sets containing the process identifiers of processes of type *Site* and *Communicator*, and *_Site* and *_Communicator* are the actions representing the transitions of these processes. Fairness conditions are generated from the fairness annotations present in the PLUSCAL model, e.g.

$$\begin{aligned} Fairness \triangleq & \wedge \forall self \in Communicator : WF_{vars}(_Communicator(self)) \\ & \wedge \forall self \in Site : \wedge WF_{vars}(enter(self)) \\ & \wedge WF_{vars}(crit(self)) \\ & \wedge WF_{vars}(exit(self)) \end{aligned}$$

and the overall specification is obtained as

$$LamportMutex \triangleq Init \wedge \square[Next]_{vars} \wedge Fairness$$

Finally, the properties and the instance sections of the PLUSCAL model are processed in order to generate the configuration file, which defines the finite-state instance and indicates the properties to be verified with TLC.

3.4 Comparison with Lamport's PlusCal

The language that we have presented in this paper was inspired by Lamport's PLUSCAL, to which it remains close in spirit, but it attempts to overcome some of the deficiencies that we have identified in section 2. We briefly comment on what we believe are the main advantages of our language.

Self-contained models. Models written for the original PLUSCAL language can express only the algorithm. All additional information, such as fairness assumptions, correctness properties or model checking constraints have to be manually

inserted into the TLA⁺ model generated by the compiler, requiring the user to understand not only TLA⁺ but also the translation. We do not expect users to understand our compiler in any detail, or even to read the generated TLA⁺ file.

Nested processes and scoped declarations. We allow for nested process declarations, and this leads to a clearer representation of the (communication) structure of algorithms. In our running example, we were able to declare the variables of each site as local variables, with two threads (the main `Site` process and the `Communicator`) accessing them. In the original PLUSCAL, one would either have two top-level process types that need to communicate via global variables (which then must be declared explicitly as arrays by the user) or insert the message-handling code between all transitions of the `Site` process. In either case, the model becomes hard to understand, contradicting the purpose of a high-level modeling language.

Unlike the original PLUSCAL, our compiler enforces proper scoping of variables, procedures, and definitions, avoiding potential errors by inadvertently accessing the variables of a different process. In future work, we plan to take advantage of this locality information in order to implement partial-order reductions for optimizing model checking.

More flexibility. As discussed in Section 3.2, the basic idea in PLUSCAL is to specify atomicity via labels. We managed to lift some of the restrictions on label placement that were present in the original PLUSCAL language, and our compiler will add labels when they are required. The user can now enforce atomicity of code blocks containing labels using the new **atomic** statement.

We also introduced several extensions, such as the **for** statement for iterating over a set, or the possibility to have several assignments to the same variable within a group of statements. On the technical side, we strived for better modularity of translation so that it becomes easier to experiment with new language primitives.

While our PLUSCAL variant retains most of the “look and feel” of the original PLUSCAL language, it does not guarantee backward compatibility. For example, programs that modify variables that are not currently in scope will be rejected by the new PLUSCAL compiler. The current version also does not provide macros that exist in Lamport’s PLUSCAL.

There are many features that we deliberately did not implement. For example, PLUSCAL does not provide primitives for message passing between processes. Distributed algorithms use many different forms of message passing (synchronous or not, lossy, duplicating, order preserving, . . .), and these are better defined in a standard library of procedures or definitions than hard-wired into the language.

4 Experiments

We have tested our language and implementation by modeling several concurrent and distributed algorithms in it and verifying them using TLC. Our experience so

Table 1. Number of states for some algorithms

Algorithm	# proc.	original PLUSCAL	our PLUSCAL
Peterson	2	37	23
FastMutex	2	2679	2679
Naimi-Trehel	3	111749	53905

far has been quite satisfactory: we found that we could represent the algorithms in a natural way and never had to touch the generated TLA⁺ models. Table 1 shows the number of (distinct) states generated by TLC for the original PLUSCAL and the new PLUSCAL models of three algorithms: Peterson’s algorithm [13], a model of which is included in the original PLUSCAL distribution, Lamport’s FastMutex algorithm [7], a model of which appears in the PLUSCAL reference manual [10], and the distributed mutual-exclusion algorithm due to Naimi and Trehel [12], which is a refinement of Lamport’s algorithm shown in Figs. 1 and 2. Models of all but the most trivial algorithms, and in particular of distributed algorithms, tend to be much clearer in our version of PLUSCAL. The numbers in Table 1 indicate that the added expressiveness does not come at the expense of lost efficiency in verification, as the state spaces generated from the new PLUSCAL models are not bigger than those for the same algorithm written in the original PLUSCAL.⁴ In some cases, we obtain smaller numbers of states because of lifted labeling restrictions. The running times of TLC for these small examples never exceeded a few seconds. In future work, we believe that we can achieve significant improvements by exploiting the information about locality in PLUSCAL for implementing partial-order reductions.

The simplicity of translation to TLA⁺ was an important design objective for the original PLUSCAL language. In particular, it is tolerable that counter-examples generated by the model checker are displayed in terms of the generated TLA⁺ model, which the user has to understand anyway. Some of our extensions, and in particular nested processes with local variables, complicate the translation and the interpretation of counter-examples. We intend to implement a filter for displaying counter-examples in terms of the original PLUSCAL model.

5 Related Work

There are many languages for modeling concurrent and distributed algorithms. PROMELA [4] is the modeling language for verification of distributed systems using the SPIN model checker. A PROMELA model consists of processes, channels and variables. PROMELA does not support nested processes, has fixed primitives for communication, and rather low-level representations of data (fixed-width subsets of integers, records, and channels). It is therefore better suited to lower-level descriptions of algorithms and protocols. On the other hand, SPIN offers more efficient verification techniques than TLC.

⁴ Moreover, handwritten TLA⁺ models of these algorithms that have comparable “grain of atomicity” generate similar numbers of states.

LOTOS [1] (Language of Temporal Ordering Specifications) is a formalism for specifying distributed systems, specifically related to Open Systems Interconnection (OSI) computer network architecture. Estelle [2] is another formal description technique for writing specification for concurrent and distributed information processing systems. It is based on Extended State Transition systems and is supported by industrial-strength tools. Both languages are similar in purpose to PROMELA, whereas we are aiming at obtaining higher-level descriptions of algorithms, for which abstract data representations in terms of sets and functions are more useful.

There exist many other languages that are closer to the programming languages rather than formal specification languages. They serve as inputs to verification tools and/or for generating executable code. For example, Mace [5] is a language for building distributed systems. It is a C++ language extension that translates distributed system specifications into a C++ implementation. Model checking can be performed at a higher level using the MaceMC model checker. In contrast, PLUSCAL is intended as a language that algorithm designers use to communicate (and validate) their ideas, not for generating efficient implementations.

6 Conclusion

PLUSCAL is a high-level language that aims at natural expression of algorithms; it makes formal verification easily accessible to algorithm designers. We have identified certain limitations of the original language and have defined a new version that tries to overcome them. In particular, we have strived at making algorithm descriptions entirely self-contained, so that knowledge of TLA⁺, and in particular of the PLUSCAL compiler, is no longer a prerequisite for using PLUSCAL. We have also made the language more uniform, removing some limitations and adding features such as nested processes, scoped declarations, and user-defined grain of atomicity. We believe that the new version significantly simplifies the representation of algorithms in PLUSCAL and that it will be more accessible to users who are not experts in formal methods.

In future work, we are planning to address reduction techniques for mitigating the effect of state space explosion. In particular, we plan to implement partial-order reduction for verifying models written in PLUSCAL. In concurrent and distributed systems, the execution of independent events in an arbitrary order results in the same overall system state, and it is therefore enough to consider only one interleaving of such events. Efficiently verifying that two events are independent is, however, non-trivial, and adding locality to PLUSCAL was an important first step in identifying sufficient conditions for two statements being independent.

On a more technical level, it would be beneficial to translate counter-examples produced by TLC back to the level of PLUSCAL programs in order to make them easier to understand for PLUSCAL users. We also plan to integrate our PLUSCAL language into the TLA⁺ toolbox that has recently been released [5].

⁵ <http://www.tlaplus.net/>

Acknowledgement. We are grateful to Leslie Lamport for discussions on the design of our variant of PLUSCAL and for his encouragement of our project.

References

1. Bolognesi, T., Brinksma, E.: Introduction to the ISO specification language LOTOS. *Computer Networks* 14 (1987)
2. Budkowski, S., Dembinski, P.: An introduction to Estelle: A specification language for distributed systems. *Comput. Netw. ISDN Syst.* 14(1), 3–23 (1987)
3. Dijkstra, E.W.: Guarded commands, non-determinacy and formal derivation of programs. *Commun. ACM* 18(8), 453–457 (1975)
4. Holzmann, G.: *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, Reading (2004)
5. Killian, C.E., Anderson, J.W., Braud, R., Jhala, R., Vahdat, A.: Mace: Language Support for Building Distributed Systems. In: *PLDI*, pp. 179–188 (2007)
6. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21(7), 558–565 (1978)
7. Lamport, L.: A fast mutual-exclusion algorithm. *ACM Trans. Computer Systems* 5(1), 1–11 (1987)
8. Lamport, L.: *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Reading (2002)
9. Lamport, L.: Checking a multithreaded algorithm with +CAL. In: Dolev, S. (ed.) *DISC 2006*. LNCS, vol. 4167, pp. 151–163. Springer, Heidelberg (2006)
10. Lamport, L.: *A +CAL user’s manual* (2007), <http://research.microsoft.com/en-us/um/people/lamport/tla/pluscal.html>
11. Lynch, N.A.: *Distributed Algorithms*. Morgan Kaufmann, San Francisco (1996)
12. Naimi, M., Trehel, M., Arnold, A.: A $\log(n)$ distributed mutual exclusion algorithm based on path reversal. *J. Parallel Distrib. Comput.* 34(1), 1–13 (1996)
13. Peterson, G.L.: Myths about the mutual exclusion problem. *Inf. Process. Lett.* 12(3), 115–116 (1981)
14. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA+ specifications. In: Pierre, L., Kropf, T. (eds.) *CHARME 1999*. LNCS, vol. 1703, pp. 54–66. Springer, Heidelberg (1999)

A Formal Environment Model for Multi-Agent Systems

Paulo Salem da Silva and Ana C.V. de Melo

University of São Paulo
Department of Computer Science
São Paulo, Brazil
salem@ime.usp.br, acvm@ime.usp.br

Abstract. Multi-agent systems are employed to model complex systems which can be decomposed into several interacting pieces called agents. In such systems, agents exist, evolve and interact within an environment. In this paper we present a model for the specification of such environments. This *Environment Model for Multi-Agent Systems* (EMMAS), as we call it, defines both structural and dynamic aspects of environments. Structurally, EMMAS connects agents by a social network, in which the link between agents is specified as the capability that one agent has to act upon another. Dynamically, EMMAS provides operations that can be composed together in order to create a number of different environmental situations and to respond appropriately to agents' actions. These features are founded on a mathematical model that we provide and that defines rigorously what constitutes an environment. Formality is achieved by employing the π -calculus process algebra in order to give the semantics of this model. This allows, in particular, a simple characterization of the evolution of the environment structure. Moreover, owing to this formal semantics, it is possible to perform formal analyses on environments thus described. For the sake of illustration, a concrete example of environment specification using EMMAS is also given.

1 Introduction

Multi-agent systems (MAS) [13] can be used to model complex systems in which the entities to be studied can be decomposed into several interacting pieces called *agents*. Human societies, computer networks, neural tissue and cell biology are examples of systems that can be seen from this perspective. Given a MAS, one technique often employed to study it is simulation [2]. That is, one may implement the several agents of interest, compose them into a MAS, and then run simulations in order to analyse their dynamic behavior. In such works, the analysis method of choice is usually the collection or optimization of statistics over several runs (e.g., the mean value of a numeric variable over time). Examples of this approach include platforms such as Swarm [5], MASON [3] and Repast [6]. There are, however, other possibilities for analysing such simulations. The crucial insight here is that simulations can be seen as incomplete explorations of state-spaces, and thus can be subject to some kinds of formal analyses.

A MAS can be decomposed into two aspects. The first relates to the agents. The second deals with how such agents come together and interact among themselves. The elements that form this second aspect constitute the *environment*¹ of a MAS.

That said, our overall work is concerned with how one can build a MAS to model a complex situation suitable for both exploratory simulation and approximate formal verification. To achieve this, we aim at providing three basic elements: (i) an agent model, which we have already described in [10]; (ii) a formal specification of the environment of these agents, so that they can be composed into a MAS; and (iii) techniques to formally analyse the resulting MAS.

In this paper we focus on the problem of defining environments. Our environments have a social network structure in which nodes are agents, and the links between them are defined by the capabilities that agents have to act upon each other. Furthermore, environments are more than a network structure, as they may change dynamically, either spontaneously or as a reaction to an agent's actions. These design choices arise from the agent model that we consider [10]. In it, agents are described from the point of view of behavioral psychology [11], which suggests a number of desirable features from an environment that brings them together. For instance, great importance is placed on the possibility of performing experiments of different kinds, and of responding to agent's actions in appropriate ways. As we shall see, our approach achieves this by the *environment behaviors* it defines. Furthermore, interaction is mostly interestingly treated by abstracting physical properties away and dealing only with relationships, which we do by adopting a social network structure and operations to modify it. We believe that these characteristics already differentiate our work substantially from other existing environment description methods (see Weyns *et al.* [14] for a survey).

Here we develop a simple formal framework in which to define such environments so that they can be subject to automated analyses procedures. A mathematical model is provided, which we call the *Environment Model for Multi-Agent Systems* (EMMAS), and its semantics is given in terms of the π -calculus process algebra [47].

Process algebras are typically employed to describe concurrent systems. They are good at succinctly describing behaviors relevant for inter-process communication. Our particular choice of π -calculus as a theoretical foundation is motivated by a number of its distinguishing features among existing such algebras. First, it takes communication through channels as a primitive notion, which makes it a natural choice for representing networks. Second, it allows for dynamic modification, which makes the creation and destruction of connections between agents possible. Third, it provides a convenient representation for broadcast behavior

¹ Notice that the term “environment” is not used consistently in the MAS literature [14]. Sometimes, it is used to mean the conceptual entity in which the agents and other objects exist and that allows them to interact; sometimes, it is used to mean the computational infrastructure that supports the MAS (e.g., a simulator). We use the term in the former sense.

through its replication operator. Finally, it has few operators and a simple operational semantics, which is attractive for implementation.

It is worth to note that despite all of these qualities of process algebras in general, and of π -calculus in particular, they are not usually employed in the context of multi-agent systems simulation. One exception is the work of Wang and Wysk [12], which uses a modified π -calculus to express a certain class of agents and their environments. But their approach is not sufficient to deal with our problems, and thus we develop our own method.

We purposefully treat agents as black-boxes here. This does not mean that they have no known internal structure; it merely means that such structure is mostly irrelevant as far as their environment is concerned. We assume, thus, that those two aspects of a MAS are complementary, but separate, issues. However, there must be a way to interface the agents with their environment. This is achieved through the assumption that agents receive *stimuli* as input and that they output *actions*.

The text is organized as follows. Section 2 introduces the basic features of the model, and also provides their semantics. Section 3, in turn, defines a number of convenience elements, which are not fundamental, but form a valuable specification repertoire. The reader is supposed to be familiar with the π -calculus process algebra, though the presented specifications are straightforward and should perhaps be accessible to anyone with some knowledge of process algebras. Section 4 presents a concrete example of an EMMAS specification. At last, Sect. 5 summarizes the main points presented and considers the new perspectives that EMMAS brings. The present text is based on and an evolution of a longer technical report [9], which the reader might wish to consult as well.

For the sake of readability, we have omitted π -calculus input and output parameters when such parameters are not relevant (e.g., we write \bar{a} instead of $\bar{a}(x)$ if x is not used later).

2 Environment Model

Our *Environment Model for Multi-Agent Systems* (EMMAS) is a mathematical framework that can be used to specify environments for multi-agent systems. Its translation to the π -calculus process algebra is achieved using a translation function to map constructs of EMMAS into π -calculus expressions (i.e., a construct C is translated to $[C]_\pi$). The full definition of such a function will be given as new constructs are introduced, and for the moment the following suffices.

Definition 1 (Translation function). *The translation function $[]_\pi$ maps constructs of EMMAS into π -calculus expressions.*

2.1 Underlying Elementary π -Calculus Events

A π -calculus specification can be divided into two parts. First, and most fundamentally, it is necessary to specify the set of events that are particular to that

specification. Second, it is necessary to specify processes built using those events. In this section we account for this first part.

Input and output events are all made from basic names. Hence, we first formally define a set of names in order to have the corresponding events. The definition below define such names, and Table 1 explains the events that arise.

Definition 2 (Environment Names). *The environment names are defined by the following set:*

$$ENames = \{emit_a^n, stop_a^n, beginning_s^n, stable_s^n, absent_s^n, destroy_{a,n}^{s,m}, ccn, done \mid a \in Actions, s \in Stimuli, m, n \in AgentIDs\}$$

Moreover, the set of environment events that immediately follow from $ENames$ is called $EEvents$.

Notice that names are primitive entities, even though they are denoted here with subscripts and superscripts, which could suggest some sort of parametrization. This writing style is merely for readability's sake.

Table 1. Informal description of events, divided in three categories according to their origin and destination. The corresponding output or input events not shown merely allow the ones described to work properly.

Event	Informal description
<i>Agent to environment</i>	
$emit_a^n$	Agent identified by n performs action a .
$stop_a^n$	Agent identified by n stops performing action a .
<i>Environment to agent</i>	
$beginning_s^n$	Delivery of stimulus s to the agent identified by n is beginning.
$stable_s^n$	Delivery of stimulus s to the agent identified by n is stable.
$ending_s^n$	Delivery of stimulus s to the agent identified by n is ending.
$absent_s^n$	Delivery of stimulus s to the agent identified by n becomes absent.
<i>Environment to environment</i>	
$destroy_{a,n}^{s,m}$	Requests the destruction of an action transformer that converts action a from agent identified by n into stimulus s accepted by the agent identified by m .
ccn	Requests the creation of a new action transformer.
$done$	Signals that an operation has terminated.

2.2 Operations

In order to exhibit dynamic behavior, the environment depends on *operations* to modify its structures.

Definition 3 (Operation). *An operation is any π -calculus expression such that:*

- *its names belong to the set $E\text{Names}$;*
- *it signals its termination with the $\overline{\text{done}}$ event.*

The second condition is particularly important because it will allow the sequential composition of operations, as we shall see in Sect. 3.1.

Of course such an abstract definition of operations cannot be used directly. Nevertheless, it suffices to define the basic model for environments. Concrete operations shall be given in Sect. 3.2.

2.3 Environment Structures

The *environment* is the central structure of EMMAS specifications. It defines which agents are present, how they are initially connected, and what dynamic behaviors exist in the environment itself. The presentation below follows a top-down approach. We begin by defining the overall environment, and then proceed to examine the nature of its constituent parts.

Definition 4 (Environment). *An environment is a tuple $\langle AG, AT, EB \rangle$ such that:*

- $AG = \{ag_1 \dots ag_l\}$ *is a set of agent profiles;*
- $AT = \{t_1 \dots t_m\}$ *is a set of action transformers;*
- $EB = \{eb_1 \dots eb_n\}$ *is a set of operations (Def. 3), which are called here environment behaviors.*

Moreover, let $E\text{Names} = \{en_1, \dots, en_o\}$. Then the corresponding π -calculus expression for the environment is defined as:

$$[\langle AG, AT, EB \rangle]_\pi = (\nu en_1, \dots, en_o) \\ ([ag_1]_\pi | [ag_2]_\pi | \dots | [ag_l]_\pi | \\ [t_1]_\pi | [t_2]_\pi | \dots | [t_m]_\pi | \\ [eb_1]_\pi | [eb_2]_\pi | \dots | [eb_n]_\pi | \\ !\text{NewAT})$$

where

$$\text{NewAT} = \text{ccn}\langle \text{emit}, \text{stop}, \text{absent}, \text{beginning}, \text{stable}, \text{ending}, \text{destroy} \rangle. \\ T(\text{emit}, \text{stop}, \text{absent}, \text{beginning}, \text{stable}, \text{ending}, \text{destroy})$$

and T is given in Def. 6.

This definition merits a few comments. First, all elements are put in parallel composition, which allows them to interact. Notice that all names from $E\text{Names}$ are restricted to the environment, which ensures that events are always used in such an interaction (i.e., events cannot be sent to outside the environment

process, and therefore can only be used internally). Second, the set of action transformers provide the network structure that connects the agents, as we shall shortly see. Third, the environment behaviors, as the name implies, specifies behaviors that belong to the environment itself. This is useful to model reactions to agents' actions, as well as to capture ways in which the environment may evolve. This is achieved through operations provided by the specifier. Finally, the component *NewAT* allows the creation of new action transformers. In order to do so, it receives a message \overline{ccn} ("create connection"), whose parameters initialize the rest of the expression. We shall see an operation that does this in Sect. 3.2.

Environments exist in order to allow agents to interact. As we remarked earlier, the internal structure of these agents, as complex as it may be, is mostly irrelevant for their interaction model. Thus, we have abstracted it away as much as possible. What is left are the interfaces that allow agents to interact with each other and with the environment itself, which we call *agent profiles*. Hence, we have the following definition.

Definition 5 (Agent Profile). *An agent profile is a triple $\langle n, S, A \rangle$ such that:*

- $n \in \text{AgentIDs}$ is a unique identifier for the agent;
- $A = \{a_1 \dots a_i\} \subseteq \text{Actions}$ is a set of actions;
- $S = \{s_1 \dots s_j\} \subseteq \text{Stimuli}$ is a set of stimuli.

Moreover,

$$\langle n, S, A \rangle_\pi = ([\text{Act}(a_1, n)]_\pi | [\text{Act}(a_2, n)]_\pi | \dots | [\text{Act}(a_i, n)]_\pi) | ([\text{Stim}(s_1, n)]_\pi | [\text{Stim}(s_2, n)]_\pi | \dots | [\text{Stim}(s_j, n)]_\pi)$$

such that, for all $a \in A$ and $s \in S$, we have:

$$[\text{Act}(a, n)]_\pi = !(\overline{\text{emit}}_a^n . \overline{\text{stop}}_a^n)$$

$$[\text{Stim}(s, n)]_\pi = \text{piStim}(\text{beginning}_s^n, \text{stable}_s^n, \text{ending}_s^n, \text{absent}_s^n)$$

where

$$\text{piStim}(\text{beginning}, \text{stable}, \text{ending}, \text{absent}) = \text{beginning} . \text{stable} . \text{ending} . \text{absent} . \text{piStim}(\text{beginning}, \text{stable}, \text{ending}, \text{absent})$$

In this definition, it is clear that agents have several components, each responsible for controlling one particular action or stimulus. $\text{Act}(a, n)$ defines that the agent identified by n can start emitting an action a and can then stop such emission. The replication operator ensures that this sequence can be carried out an unbounded number of times. $\text{Stim}(s, n)$, in turn, defines that the agent identified by n can be stimulated by s , and that this stimulation follows four steps (i.e., *absent*, *beginning*, *stable* and finally *ending*). The recursive call ensures

that this stimulation sequence can start again as soon as it finishes the last step. These definitions reflect the assumptions about the agent model we consider [10], which, in particular, defines precise – internal – consequences for each of these stimulation steps.

Agents interact by stimulating each other. But to have this capability, it is first necessary to define that an agent’s action causes a stimulation in another agent. This is done through *action transformers*, which specifies that if agent ag_1 performs the action a , then agent ag_2 should be stimulated with s .

Definition 6 (Action Transformer). *An action transformer is a tuple $\langle ag_1, a, s, ag_2 \rangle$ such that:*

- ag_1 is an agent profile $\langle n, S_1, A_1 \rangle$;
- ag_2 is an agent profile $\langle m, S_2, A_2 \rangle$;
- a is an action such that $a \in A_1$;
- s is a stimulus such that $s \in S_2$;

Moreover, the corresponding π -calculus expression for the action transformer is defined as:

$$[\langle ag_1, a, s, ag_2 \rangle]_{\pi} = T(\text{emit}_a^n, \text{stop}_a^n, \text{absent}_s^m, \text{beginning}_s^m, \text{stable}_s^m, \text{ending}_s^m, \text{destroy}_{a,n}^{s,m})$$

where

$$T(\text{emit}, \text{stop}, \text{absent}, \text{beginning}, \text{stable}, \text{ending}, \text{destroy}) = \underbrace{(\text{emit}.\text{beginning}.\text{stable}.\text{stop}.\text{ending}.\text{absent}.)}_{\text{Normal behavior}} + \underbrace{T(\text{emit}, \text{stop}, \text{absent}, \text{beginning}, \text{stable}, \text{ending}, \text{destroy})}_{\text{destroy}}$$

To disable the action transformer

The above definition can be divided in two parts. First, there is its normal behavior, which merely defines the correct sequence through which an action is transformed in a stimulus. Once such a sequence is completed, a recursive call to the process definition restarts the action transformer. Second, there is the part that allows the transformer to be destroyed. By performing *destroy*, the action transformer disappears, since this event is not followed by anything. Figure 1 shows an example of environment in which the role of action transformers can be appreciated.

We choose to have an intermediate structure such as the action transformer between the agents instead of allowing a direct communication because an agent’s actions may have other effects besides stimulation. In particular, the environment can also respond to such actions in custom ways through the specified environment behaviors.

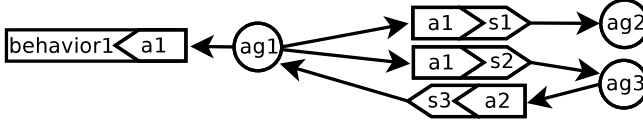


Fig. 1. An example of environment. Circles denote agent profiles ag_1 , ag_2 and ag_3 . There are three action transformers: $\langle ag_1, a_1, s_1, ag_2 \rangle$, $\langle ag_1, a_1, s_2, ag_3 \rangle$ and $\langle ag_3, a_2, s_3, ag_1 \rangle$. Moreover, there is also an environment behavior, $behavior_1$, that is executed whenever agent ag_1 performs action a_1 . Notice that the same action a_1 , performed by agent ag_1 , has three simultaneous consequences. Notice further that while two of these consequences are stimulations, another merely triggers some operation. This shows that it is technically interesting to have actions and stimuli as different entities, since they are not always related.

2.4 Semantics

The semantics of EMMAS is given by considering: (i) a syntactical translation of EMMAS into π -calculus expressions; and (ii) a mathematical foundation which relates π -calculus events to the stimuli and actions of agents. The π -calculus translation of (i), through its operational semantics, provides an over-approximation of the desired behavior, which is then made precise using the restrictions provided by (ii). By this method, we shall be able to build an LTS that defines the possible states and transitions for any particular environment specification.

For the sake of clarity, we divide this section in two parts. First we define some preliminary structures required for building the transition system, which is then presented.

Preliminary Definitions. Our model must have a way to effectively interact with the agents of a MAS. Agents may trigger events that have a meaning in the environment specification (e.g., the performance of an action). Conversely, the environment specification may request the performance of an operation (e.g., to stimulate an agent). It is necessary, therefore, to have a mathematical foundation that formally defines how to accomplish this. We fulfill this requirement by providing both a *vocabulary* in which a few primitives are defined and a definition for what constitutes an *environment status* with respect to these primitives.

Definition 7 (Vocabulary). A vocabulary is a tuple

$$\langle Stimuli, Actions, AgentIDs \rangle$$

such that:

- *Stimuli* is a finite set of stimuli;
- *Actions* is a finite set of actions;
- *AgentIDs* is a finite set of agent identifiers;

The sets *Stimuli*, *Actions* and *AgentIDs* define, respectively, all available stimuli, actions and agent identifiers. These are sets containing primitive, unstructured, elements.

Definition 8 (Environment Status)

An environment status is a pair

$$\langle \textit{Stimulation}, \textit{Response} \rangle$$

such that:

- *Stimulation* : $\textit{AgentIDs} \times \textit{Stimuli} \rightarrow \{\textit{Beginning}, \textit{Stable}, \textit{Ending}, \textit{Absent}\}$;
- *Response* : $\textit{AgentIDs} \times \textit{Actions} \rightarrow \{\textit{Emitting}, \textit{NotEmitting}\}$.

Building the Transition System. Given an environment E , we shall build an *annotated environment LTS* by considering the LTS induced by $[E]_\pi$, whose states shall be annotated with our environment status (Def. 8), and whose structure shall be subject to some restrictions based on the possible values for an environment status. Then we shall then have an LTS whose states have the following form.

Definition 9 (State). Let E be an environment and P be a π -calculus process obtained by applying π -calculus operational semantics rules to $[E]_\pi$. Moreover, let $\langle \textit{Stimulation}, \textit{Response} \rangle$ be an environment status. Then a state is defined as the following pair:

$$(P, \langle \textit{Stimulation}, \textit{Response} \rangle)$$

By this construction, at any point of the LTS we shall be able to know both what is the current situation of the agents (because of the added environment status) and what are the possible changes from that point (because of the π -calculus operational semantics).

To proceed with this construction, we need a number of definitions. Let us begin by providing a way to observe the internal transitions of an environment, which is a fundamental capability that we need before proceeding. Recall from Def. 4 that an environment's π -calculus process has a number of restrictions that would prevent such observations (i.e., the transitions would be internal to the process and not discernible in the LTS). It is, however, possible to characterize these restrictions syntactically, and thus we may provide a simple method to remove them when needed. This is accomplished by the following *environment unrestriction function* unr .

Definition 10 (Environment Unrestriction Function)

Let P and Q be π -calculus processes such that

$$P = (\nu \textit{en}_1, \dots, \textit{en}_o)Q$$

where $\{\textit{en}_1, \dots, \textit{en}_o\} = \textit{ENames}$. Then the environment unrestriction function is defined as $unr(P) = Q$.

We may now define the *Stimulation* function present in each state as follows.

Definition 11 (Stimulation)

Let $(P, \langle \textit{Stimulation}, \textit{Response} \rangle)$ be a state. Moreover, let \rightarrow be the transition relation induced by the π -calculus operational semantics. Then, for all $s \in \textit{Stimuli}$ and $n \in \textit{AgentIDs}$, we have:

$$\textit{Stimulation}(n, s) = \begin{cases} \textit{Absent} & \textit{if } \exists P' \textit{ such that } \textit{unr}(P) \xrightarrow{\textit{beginning}_s^n} P' \\ \textit{Beginning} & \textit{if } \exists P' \textit{ such that } \textit{unr}(P) \xrightarrow{\textit{stable}_s^n} P' \\ \textit{Stable} & \textit{if } \exists P' \textit{ such that } \textit{unr}(P) \xrightarrow{\textit{ending}_s^n} P' \\ \textit{Ending} & \textit{if } \exists P' \textit{ such that } \textit{unr}(P) \xrightarrow{\textit{absent}_s^n} P' \end{cases}$$

The *Stimulation* definition establishes the status of a particular stimulation based on the order that stimulations must change (see Def. 5). For instance, if a process is capable of receiving a *beginning*_sⁿ event, it must be the case that stimulus s is currently absent in agent identified by n . The *Stimulation* function, therefore, merely gives a way of reading the π -calculus LTS in order to have this information explicitly for every agent and stimulus in any given process.

The *Response* function, on the other hand, is assumed as given (e.g., by a simulator that implements the black-box behavior of the agents). Thus, we do not define it. However, it imposes some constraints on the LTS, which we must specify and take in account. As we shall see shortly, these constraints turn the π -calculus over-approximation into an exact description of the transition system's structure that we wish to assign to EMMAS.

Definition 12 (Transition constraints)

Let $s_1 = (P_1, \langle \textit{Stimulation}_1, \textit{Response}_1 \rangle)$ and $s_2 = (P_2, \langle \textit{Stimulation}_2, \textit{Response}_2 \rangle)$ be states in an annotated environment LTS $\langle S, L, \rightsquigarrow \rangle$. Moreover, let \rightarrow be the transition relation induced by the π -calculus operational semantics. Then the transition $s_1 \xrightarrow{!} s_2$ is forbidden if one of the cases hold:

- there exists $a \in \textit{Actions}$ and $n \in \textit{AgentIDs}$ such that:
 - $\textit{Response}_1(n, a) = \textit{Emitting}$;
 - P_2 was obtained by internally producing the event $\overline{\textit{stop}}_a^n$ in P_1 .
- there exists $a \in \textit{Actions}$ and $n \in \textit{AgentIDs}$ such that:
 - $\textit{Response}_1(n, a) = \textit{NotEmitting}$;
 - P_2 was obtained by internally producing the event $\overline{\textit{emit}}_a^n$ in P_1 .
- there exists $a \in \textit{Actions}$ and $n \in \textit{AgentIDs}$ such that:
 - $\textit{Response}_1(n, a) = \textit{Emitting}$;
 - $\textit{Response}_2(n, a) = \textit{NotEmitting}$;
 - there exists a P' such that $\textit{unr}(P_1) \xrightarrow{\textit{emit}_a^n} P'$.
- there exists $a \in \textit{Actions}$ and $n \in \textit{AgentIDs}$ such that:
 - $\textit{Response}_1(n, a) = \textit{NotEmitting}$;
 - $\textit{Response}_2(n, a) = \textit{Emitting}$;
 - there exists a P' such that $\textit{unr}(P_1) \xrightarrow{\textit{stop}_a^n} P'$.

At last, we may define the annotated environment LTS as follows.

Definition 13 (Annotated Environment LTS). *Let E be an environment (Def. 4), and let \rightarrow be the transition relation induced by the π -calculus operational semantics. Then an annotated environment LTS is an LTS $\langle S, L, \rightsquigarrow \rangle$ such that:*

- $L = E\text{Events}$ (see Def. 2);
- S and \rightsquigarrow are constructed inductively as follows:
 - **Initial state.** $([E]_\pi, es) \in S$, where $es = \langle \text{Stimulation}, \text{Response} \rangle$ such that for all $a \in \text{Actions}$, $s \in \text{Stimuli}$, and $n \in \text{AgentIDs}$ we have $\text{Stimulation}(n, s) = \text{Absent}$ and $\text{Response}(n, a) = \text{NotEmitting}$.
 - **Other states and transitions.**
 If $s_1 = (P_1, \langle \text{Stimulation}_1, \text{Response}_1 \rangle) \in S$,
 then $s_2 = (P_2, \langle \text{Stimulation}_2, \text{Response}_2 \rangle) \in S$ and $s_1 \xrightarrow{l} s_2$ if and only if:
 - * $P_1 \xrightarrow{l} P_2$;
 - * Stimulation_2 is defined w.r.t. P_2 according to Def. 11;
 - * $s_1 \rightsquigarrow s_2$ is not forbidden by Def. 12.

3 Convenience Elements and Operations

So far we have defined the bare minimum for describing environments so that they can be formally analysed. Clearly, though, more constructs are necessary in order to make such specifications. For example, in Def. 3 we established what is an operation in general, but we have not presented any particular one. In the present section, then, we provide a number of convenience elements that can be used to build concrete EMMAS models. These, however, are merely examples of what can be expressed with the basic model given before, designed to show its usefulness, and the reader may well imagine many other convenience elements.

3.1 Composition Operators

In order to build complex operations on top of the basic ones, it is useful to define composition operators. Some of these can be mapped directly to π -calculus operators, but others require more sophistication.

Definition 14 (Sequential Composition). *Let Op_1 and Op_2 be operations. Then their sequential composition is also an operation and is written as:*

$$Op_1; Op_2$$

Moreover,

$$[Op_1; Op_2]_\pi = (\nu \text{ start})[Op_1]_\pi \{ \text{start/done} \} | \text{start}. [Op_2]_\pi$$

The above translation aims at accounting for the intuition that Op_1 must take place before Op_2 . However, we cannot translate $Op_1;Op_2$ immediately as $[Op_1]_\pi.[Op_2]_\pi$, because in general π -calculus would not allow the resulting syntax (e.g., $(P + Q).R$ would not be a valid expression). Therefore, we adapt the suggestion offered by Milner [4] (in Example 5.27), which requires every operation to signal its own termination with a *done* event.

Definition 15 (Sequence). *Let Op be an operation and n be an integer such that $n \geq 1$. Then a sequence of n compositions of Op is defined as:*

$$Seq(Op, n) = \begin{cases} Op; Seq(Op, n - 1) & n > 1 \\ Op & n = 1 \end{cases}$$

Definition 16 (Unbounded Sequence). *Let Op an operation. Then an unbounded sequence of compositions of Op is defined as:*

$$Forever(Op) = Op; Forever(Op)$$

The translation of these two kinds of sequences to π -calculus follows, of course, from the translation of the sequential composition operator.

Definition 17 (Choice). *Let Op_1 and Op_2 be operations. Then their composition as a choice is also an operation and is written as:*

$$Op_1 + Op_2$$

Moreover,

$$[Op_1 + Op_2]_\pi = [Op_1]_\pi + [Op_2]_\pi$$

Definition 18 (Parallel Composition). *Let Op_1, Op_2, \dots, Op_n be n operations. Then their parallel composition is also an operation and is written as:*

$$Op_1 \parallel Op_2 \parallel \dots \parallel Op_n$$

Moreover,

$$[Op_1 \parallel Op_2 \parallel \dots \parallel Op_n]_\pi = (\nu \text{ start})[Op_1]_\pi\{\text{start}/\text{done}\} | [Op_2]_\pi\{\text{start}/\text{done}\} | \dots | [Op_n]_\pi\{\text{start}/\text{done}\} | \underbrace{\text{start.start} \dots \text{start}}_{n \text{ times}} . \text{done}$$

The translation for the parallel composition is not straightforward because it is necessary to ensure that *done* is sent only once in the composed operation. That is to say, the parallel composition of n operations² is an operation itself, and it only terminates when each of its components terminates.

3.2 Core Operations

We can now provide a core of operations upon which others can be built.

² We define the operator for n operations instead of just two because this avoids the problem of establishing its associativity properties.

Agent Stimulation Operations. The following operations are provided to control the stimulation of agents.

Definition 19 (Begin stimulation operation). *Let $ag = \langle n, S, A \rangle$ be an agent profile, and $s \in S$ be a stimulus. Then the begin stimulation operation is written as:*

$$\text{BeginStimulation}(s, ag)$$

Moreover,

$$[\text{BeginStimulation}(s, ag)]_\pi = \overline{\text{beginning}_s^n \cdot \text{stable}_s^n \cdot \text{done}}$$

Definition 20 (End stimulation operation). *Let $ag = \langle n, S, A \rangle$ be an agent profile, and $s \in S$ be a stimulus. Then the end stimulation operation is written as:*

$$\text{EndStimulation}(s, ag)$$

Moreover,

$$[\text{EndStimulation}(s, ag)]_\pi = \overline{\text{ending}_s^n \cdot \text{absent}_s^n \cdot \text{done}}$$

Definition 21 (Stimulate operation). *Let $ag = \langle n, S, A \rangle$ be an agent profile, and $s \in S$ be a stimulus. Then the stimulate operation is defined as:*

$$\text{Stimulate}(s, ag) = \text{BeginStimulation}(s, ag); \text{EndStimulation}(s, ag)$$

Action Transformers Operations. The following operations are provided to manipulate action transformers.

Definition 22 (Create action transformer operation). *Let $ag_1 = \langle n, S_1, A_1 \rangle$ be an agent profile, $ag_2 = \langle m, S_2, A_2 \rangle$ be another agent profile, $a \in A_1$ be an action, and $s \in S_2$ be a stimulus. Then the create action transformer operation is written as:*

$$\text{Create}(ag_1, a, s, ag_2)$$

Moreover,

$$[\text{Create}(ag_1, a, s, ag_2)]_\pi = \overline{\text{ccn}}(\text{emit}_a^n, \text{stop}_a^n, \text{absent}_s^m, \text{beginning}_s^m, \text{stable}_s^m, \text{ending}_s^m, \text{destroy}_{a,n}^{s,m}).\overline{\text{done}}$$

In the above definition, notice that $\overline{\text{ccn}}$ is crafted to react with the component *NewAT* given in Def. 4. Since operations will ultimately be put together with parallel composition in the environment, it follows that the $\text{Create}(ag_1, a, s, ag_2)$ operation will be able to react with *NewAT* and originate a new action transformer.

Definition 23 (Destroy action transformer operation). *Let $ag_1 = \langle n, S_1, A_1 \rangle$ be an agent profile, $ag_2 = \langle m, S_2, A_2 \rangle$ be another agent profile, $a \in A_1$ be an action, and $s \in S_2$ be a stimulus. Then the destroy action transformer operation is written as:*

$$\text{Destroy}(n, a, s, m)$$

Moreover,

$$[\text{Destroy}(n, a, s, m)]_\pi = \overline{\text{destroy}_{a,n}^{s,m}}.\overline{\text{done}}$$

4 Example

Let us consider the following simple example. We shall specify an online social network, in which users may register themselves and interact.³ The objective of the specification is to test advertisement strategies through simulation. To this end, we define an environment $\langle AG, AT, EB \rangle$ with n agents, such that each agent $ag_i \in AG$ is capable of performing the action *buy* (i.e., to buy the advertised product) and *sendMsg* (i.e., to send some message to another agent), as well as receiving the stimuli gui_1, gui_2 (i.e., the graphical user interface of the website can be set in two different ways), ad_1, ad_2 (i.e., there are two different possible advertisements) and *msg* (i.e., a message received). Formally, $ag_i = \langle i, \{gui_1, gui_2, ad_1, ad_2, msg\}, \{buy, sendMsg\} \rangle$.

The action transformers among the agents allow them to send messages to their friends. Of course, the particular topology of this network can vary, but in essence each agent ag_i shall have some action transformers of the form $\langle ag_i, sendMsg, msg, ag_j \rangle$. The effect of such a message could be similar to that of an advertisement (i.e., a product recommendation by a friend).

Finally, and most importantly, for each agent $ag \in AG$, we define the following new environment behavior $eb_i \in EB$:

$$\begin{aligned} & (BeginStimulation(gui_1, ag_i) + BeginStimulation(gui_2, ag_i)); \\ & (Stimulate(ad_1, ag_i) + Stimulate(ad_2, ag_i)) \end{aligned}$$

These eb_i specify four possible simulation sequences concerning each agent. For instance, in some simulation run, agent ag_1 could be stimulated by $BeginStimulation(gui_1, ag_1)$ and then by $Stimulate(ad_1, ag_1)$. However, it could be that this particular sequence would be ineffective in eliciting the agent's *buy* action, in which case another sequence could be tried. The important thing, though, is that these trials can all be performed automatically, since they are explicit in the environment definition. This shows how EMMAS can endow simulators with some formal verification capabilities.

5 Conclusion

In this paper we have presented a formalization for environments of MASs. We provided a high-level description for this formalization, with a semantics given using the π -calculus. We found necessary to perform some adjustments on the standard behavior induced by the π -calculus' operational semantics in order to allow its integration with the remaining parts of the proposed approach. Furthermore, we avoided explicit temporal references in this formalization. However, it should be possible to add an explicit notion of time to EMMAS, though this would introduce new complications as well.

The presented environments have both structural and operational aspects. That is to say, they represent certain structures, which can then be changed by

³ Actual examples of such networks include popular websites such as www.facebook.com, www.orkut.com and www.myspace.com.

certain operations. These operations serve to two purposes. First, they provide a way to specify behaviors of the environments themselves (e.g., environment responses to the actions of agents). Second, they allow the succinct specification of several possible scenarios for an environment (e.g., several possible ways of stimulating agents). This latter possibility is one of the great advantages offered by the use of a process algebra as a semantic basis (e.g., an algebraic expression $a + b$ defines the non-deterministic *possibility* of either a or b), and renders our approach particularly unique insofar as environments for MASs are concerned. We may now formulate questions concerning the analysis of our MASs:

- Since the semantics of EMMAS is given as an LTS, it follows that now we need criteria for selecting paths in it. With such paths, we shall be able to perform concrete simulations.
- Concerning implementation, we believe that the π -calculus base can be particularly useful, since we could implement its few elements in order to have our whole model on top of it. A similar approach is taken by Wang and Wysk [12]. More generally, there are programming languages based on π -calculus, such as the Join-Calculus [1] and Pict [8].

Finally, though EMMAS is designed to work with a particular agent model [10], it actually imposes few restrictions on the agents, and its principles are general. Hence, it could perhaps be adapted to work with other agent models.

Acknowledgements

The authors would like to thank Prof. Dr. Marie-Claude Gaudel (Laboratoire de Recherche en Informatique, Université Paris-Sud 11) for her numerous comments and suggestions during the preparation of this work.

This project benefited from the financial support of *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior* (CAPES) and *Conselho Nacional de Desenvolvimento Científico e Tecnológico* (CNPq).

References

1. Fournet, C., Gonthier, G.: The join calculus: A language for distributed mobile programming. In: Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.) APPSEM 2000. LNCS, vol. 2395, pp. 268–332. Springer, Heidelberg (2002)
2. Gilbert, N., Bankers, S.: Platforms and methods for agent-based modeling. Proceedings of the National Academy of Sciences of the United States 99(supplement 3) (2002)
3. Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K.: MASON: A new multi-agent simulation toolkit (2004), <http://cs.gmu.edu/~eclab/projects/mason/>
4. Milner, R.: Communicating and Mobile Systems: the Pi-Calculus. Cambridge University Press, Cambridge (1999)
5. Minar, N., Burkhart, R., Langton, C., Askenazi, M.: The Swarm simulation system: A toolkit for building multi-agent simulations (1996), working Paper 96-06-042

6. North, M., Collier, N., Vos, J.R.: Experiences creating three implementations of the Repast agent modeling toolkit. *ACM Transactions on Modeling and Computer Simulation* 16(1), 1–25 (2006), <http://repast.sourceforge.net/>
7. Parrow, J.: An introduction to the pi-calculus. In: Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.) *Handbook of Process Algebra*, pp. 479–543. Elsevier, Amsterdam (2001)
8. Pierce, B.C., Turner, D.N.: Pict: A programming language based on the pi-calculus. In: Plotkin, G., Stirling, C., Tofte, M. (eds.) *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, Cambridge (1997)
9. da Silva, P.S.: An environment specification language for multi-agent systems, Technical Report 1531 – Université Paris-Sud 11, Laboratoire de Recherche en Informatique (2009)
10. da Silva, P.S., de Melo, A.C.V.: A simulation-oriented formalization for a psychological theory. In: Dwyer, M.B., Lopes, A. (eds.) *FASE 2007*. LNCS, vol. 4422, pp. 42–56. Springer, Heidelberg (2007)
11. Skinner, B.F.: *Science and Human Behavior*. The Free Press, New York (1953)
12. Wang, J., Wysk, R.A.: A pi-calculus formalism for discrete event simulation. In: *WSC 2008: Proceedings of the 40th Conference on Winter Simulation*, Miami, Florida, pp. 703–711 (2008)
13. Weiss, G. (ed.): *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT Press, Cambridge (1999)
14. Weyns, D., Van Dyke Parunak, H., Michel, F., Holvoet, T., Ferber, J.: Environments for multiagent systems: State-of-the-art and research challenges. In: Weyns, D., et al. (eds.) *E4MAS 2004*. LNCS (LNAI), vol. 3374, pp. 1–47. Springer, Heidelberg (2005)

A Modal Interface Theory with Data Constraints

Sebastian S. Bauer¹, Rolf Hennicker¹, and Michel Bidoit²

¹ Ludwig-Maximilians-Universität München, Germany

² Laboratoire Spécification et Vérification,
CNRS & ENS de Cachan, France

Abstract. For the design of component-based software, the behavioral specification of component interfaces is crucial. We propose an extension of the theory of modal I/O-transition systems by Larsen et al. to cope with both control flow *and* data states of reactive components at the same time. In our framework, transitions model incoming or outgoing operation calls which are constrained by pre- and postconditions expressing the mutual assumptions and guarantees of the receiver and the sender of a message. We define a new interface theory by adapting synchronous composition, modal refinement and modal compatibility to the case of modal I/O-transition systems with data constraints. We show that in this formalism modal compatibility is preserved by refinement and modal refinement is preserved by composition which are basic requirements for any interface theory.

1 Introduction

A rigorous discipline of component-based software development relies strongly on interface specifications which describe the observable behavior of components [6]. Thereby, behavior is often understood from a control-flow oriented perspective considering the sequences of actions a component can perform when interacting with its environment. Of equal importance is, however, the aspect of changing data – owned by a component – when certain actions of the component are performed. We claim that the integrated treatment of control flow and data change in the context of concurrent, reactive components is not yet sufficiently understood and therefore needs further investigation.

Building on previous work on the semantics of behavior protocols for components with data states [3] we propose an interface theory on the basis of modal I/O-transition systems (MIOs) introduced in [9]. A particular advantage of modal transition systems is that they distinguish between “may” and “must” transitions which leads to a powerful refinement notion [11]: the may-transitions determine which actions are permitted in a refinement while the must-transitions specify which actions must be present in a refinement and hence in any implementation. In this way it is possible to provide abstract, loose specifications in terms of may-transitions and to fix in a stepwise way the must-transitions until an implementation, represented by a MIO with must-transitions only, is reached. Another aspect which can be conveniently formalized with modal I/O-transition

systems concerns the compatibility of interacting components: whenever an interface specification allows that a message *may* be issued, then the communication partner should be in a (control) state where it *must* be able to accept the message [4].

In this paper we extend MIOs by taking into account the specification of data constraints which enhance transitions with pre- and postconditions describing the admissible data states of a component before and after the execution of an operation. We distinguish, like in MIOs, between input, output and internal messages and, additionally, between provided, required and internal state variables. Provided and internal state variables are local to a component and describe the data states a component can adopt. In contrast to the internal state variables, provided state variables are visible to the user of a component. Required state variables belong also to the interface specification of a component, however, they are not related to the data states of the component itself but to the data states the component can observe in its environment. On this basis we study (synchronous) composition, refinement and compatibility of modal I/O-transition systems with data constraints (MIODs). In addition to relationships between control states, we take special care of the relationships between data constraints in all these cases. For instance, considering compatibility, the condition concerning control flow compatibility is extended to take into account data states: the caller of an operation must ensure that the precondition of the operation provided by the callee is satisfied and, conversely, the callee must guarantee that after the execution of the operation the postcondition expected by the caller holds. Thus, the compatibility notion takes into account the mutual assumptions and guarantees of communicating components guided by the idea that specifications provide contracts which must match when components are composed. Our main result shows that our framework satisfies the basic requirements of an interface theory: refinement is compositional and compatibility is preserved by refinement. Thus modal I/O-transition systems with data constraints support reusability and independent implementability of components via interface specifications.

Related Work. Specifications of control flow and of changing data states are often considered separately from each other. While transition systems are a popular formalism to specify the temporal ordering of messages, invariants and pre- and postconditions are commonly used to specify the effects of operations w.r.t. data states. Though approaches like CSP-OZ [8] offer means to specify both aspects, they still lack a fully integrated treatment since their expressive power is limited to cases where the effect of an operation on data states must be independent of the control-flow behavior of a component. Other related approaches are based on symbolic transition systems (STS) [7,1] but STS are mainly focussing on model checking and not on interface theories supporting the (top down) development of concurrent systems by refinement. Most closely related to our work is the study of Mouelhi et al. [12] who consider an extension of the theory of interface automata [6] to data states. Their approach does, however, not take into account modalities of transitions and modal refinements. There is also no discrimination

of different types of state variables (provided, required and internal) which, in our case, is a methodologically important ingredient to express the contract principle between interface specifications. In our previous work, we have proposed in [3] a formal semantics of behavior protocols based on model classes, but [3] does not consider modalities and a refinement notion between specifications. On the other hand, we have investigated in [4] various kinds of modal interface theories with refinements but without taking into account constraints on data states.

Outline. The paper is organized as follows. In Sect. 2 we introduce modal I/O-transition systems with data constraints (MIODs). Their refinement is considered in Sect. 3, and in Sect. 4 the composition and the compatibility of MIODs is defined. Moreover, we show in Sect. 4 that our framework satisfies the requirements of an interface theory concerning compositionality of refinement and preservation of compatibility by refinement. In Sect. 5 we finish with some concluding remarks.

Example 1. We use a *hexapod robot* as a running example. This insect-like, six-leg robot exhibits a non-trivial control-flow behavior with a significant impact of data states. When specifying such a complex subject, non-trivial synchronization and coordination problems arise which must be addressed in a behavioral specification. For the illustration of the formal definitions and concepts introduced hereafter, we will focus here only on the locomotion aspect of the robot's legs. We assume a simple component architecture: The locomotion of each leg is coordinated by a controller component, called *LegC*, and the leg motorization is wrapped in another component, named *Leg*. ■

2 Modal I/O-Transition Systems with Data Constraints

Modal I/O-transition systems (MIOs) have been introduced in [9,11] as a formalism to describe the behavior of reactive, concurrent components which interact via matching input and output actions. MIOs distinguish between “may” and “must” transitions where the former specify which transitions are allowed in a refinement while the latter specify which transitions are mandatory (i.e. must be preserved) in any refinement. Formally, a MIO S is given by a tuple $S = (states_S, start_S, acts_S, \Delta_S^{may}, \Delta_S^{must})$ of a set of states $states_S$, the initial state $start_S \in states_S$, a set of actions $acts_S = act_S^{in} \uplus act_S^{out} \uplus act_S^{int}$ being the disjoint union of sets of input, output and internal actions respectively, a may-transition relation $\Delta_S^{may} \subseteq states_S \times act_S \times states_S$, and a must-transition relation $\Delta_S^{must} \subseteq \Delta_S^{may}$. To deal with data constraints, we want to be able to specify pre- and postconditions for the actions on modal transitions. Hence, we must use more involved labels than simple action names.

Operations. Instead of actions, we consider *operations* which may have formal parameters. An operation op is of the form $opname(Par)$ where Par is a (possibly empty) set of formal parameters. We write $par(op)$ to refer to the formal parameters of an operation op . An *I/O-operation signature* $\mathcal{O} = \mathcal{O}^{prov} \uplus \mathcal{O}^{req} \uplus \mathcal{O}^{int}$

consists of pairwise disjoint sets \mathcal{O}^{prov} of *provided* operations (for inputs), \mathcal{O}^{req} of *required* operations (for outputs), and \mathcal{O}^{int} of *internal* operations. Provided operations are offered by a component and can be invoked by the environment; required operations are required from the environment and can be called by the component. To indicate that $op \in \mathcal{O}^{prov}$ we often write $?op$ and to indicate that $op \in \mathcal{O}^{req}$ we often write $!op$.

State variables. In order to equip operations with pre- and postconditions concerning the operations' formal parameters *and* the data states of a component, we must first provide a formal notation for data states. For this purpose we use state variables of different kinds which all belong to a global set SV of state variables. *Provided* state variables describe the externally visible data states, while *internal* state variables describe the hidden data states of a component. Provided and internal state variables together model the possible data states a component can adopt. There is, however, still a third kind of state variable which we call *required* state variable. Required state variables are used to refer to the data states a component expects to be visible in its environment. Formally, an *I/O-state signature* $\mathcal{V} = \mathcal{V}^{prov} \uplus \mathcal{V}^{req} \uplus \mathcal{V}^{int}$ consists of pairwise disjoint sets \mathcal{V}^{prov} , \mathcal{V}^{req} , and \mathcal{V}^{int} of provided, required and internal state variables, respectively.

Definition 1 (I/O-Signature). *An I/O-signature is a pair $\Sigma = (\mathcal{V}, \mathcal{O})$ consisting of an I/O-state signature \mathcal{V} and an I/O-operation signature \mathcal{O} .*

Predicates on states. We assume given a global set LV of logical variables, disjoint from the state variables SV such that, for each operation op , $par(op) \subseteq LV$. Moreover, we assume a set $\mathcal{S}(SV, LV)$ of *state predicates* φ and a set $\mathcal{T}(SV, LV)$ of *transition predicates* π with associated sets $var_{state}(\varphi) \subseteq SV$ of state variables and $var_{log}(\varphi) \subseteq LV$ of logical variables; analogously for π . State predicates refer to single states and transition predicates to pairs of states (pre- and poststates). For each $\mathcal{W} \subseteq SV$ and $X \subseteq LV$ we define

$$\begin{aligned} \mathcal{S}(\mathcal{W}, X) &= \{\varphi \in \mathcal{S}(SV, LV) \mid var_{state}(\varphi) \subseteq \mathcal{W}, var_{log}(\varphi) \subseteq X\}, \\ \mathcal{T}(\mathcal{W}, X) &= \{\pi \in \mathcal{T}(SV, LV) \mid var_{state}(\pi) \subseteq \mathcal{W}, var_{log}(\pi) \subseteq X\}. \end{aligned}$$

We require that both sets, $\mathcal{S}(\mathcal{W}, X)$ and $\mathcal{T}(\mathcal{W}, X)$, are closed under the usual logical connectives, like \wedge , \Rightarrow , etc. We assume that state predicates $\varphi \in \mathcal{S}(\mathcal{W}, X)$ and transition predicates $\pi \in \mathcal{T}(\mathcal{W}, X)$ are both equipped with a *satisfaction relation* $\models \varphi$ and $\models \pi$, resp., expressing universal validity of predicates w.r.t. some semantic domain of states and w.r.t. a domain of valuations for the logical variables. We will not go into further details here, but the reader may assume a predefined data universe \mathcal{U} such that concrete data states are functions mapping state variables to values in \mathcal{U} and, similarly, valuations are mappings from logical variables to values in \mathcal{U} . Then state predicates should be interpreted w.r.t. a single data state and a valuation of the logical variables, while transition predicates should be interpreted w.r.t. two data states (pre- and poststates) and a valuation of the logical variables. Universal validity $\models \varphi$ of a state predicate $\varphi \in \mathcal{S}(\mathcal{W}, X)$ then means that φ is valid for all data states, modeled by functions $\sigma : \mathcal{W} \rightarrow \mathcal{U}$,

and for all valuations $\rho : X \rightarrow \mathcal{U}$ while universal validity $\models \pi$ of a transition predicate $\pi \in \mathcal{T}(\mathcal{W}, X)$ means that π is valid for any two data states and any valuation.

The above definitions are generic and sufficient for the following considerations. Therefore, we do not fix a particular syntax for signatures and predicates here, neither a particular definition of the satisfaction relation. We claim that our notions could be easily instantiated in the context of a particular assertion language. How this would work in the case of the Object Constraint Language OCL is shown in [5].

Example 2. We exemplify the use of signatures in our running example. The component *Leg* has the I/O-signature $\Sigma_{Leg} = (\mathcal{V}_{Leg}, \mathcal{O}_{Leg})$ where

$$\begin{aligned} \mathcal{V}_{Leg}^{prov} &= \{maxStep, currStep\} & \mathcal{O}_{Leg}^{prov} &= \{init(), lift(), swing(a), drop(), retract()\} \\ \mathcal{V}_{Leg}^{req} &= \{\} & \mathcal{O}_{Leg}^{req} &= \{update(a)\} \\ \mathcal{V}_{Leg}^{int} &= \{steps\} & \mathcal{O}_{Leg}^{int} &= \{\} \end{aligned}$$

Leg has as provided state variables *maxStep*, which models the maximal step size, and *currStep* for the current step size of the leg. The only internal state variable is *steps* which counts the number of steps the leg has made since initialization (provided operation *init()*). The provided operations also include operations for the different kinds of leg motion, i.e. *lift()*, *swing(a)*, *drop()*, and *retract()*; the only required operation is *update(a)* which, after a step has made, informs the leg controller component *LegC* about the actual step size. A $(\mathcal{V}_{Leg}^{prov} \cup \mathcal{V}_{Leg}^{int})$ -data state can be given, for instance, by the function $\sigma : (\mathcal{V}_{Leg}^{prov} \cup \mathcal{V}_{Leg}^{int}) \rightarrow \mathcal{U}$ with $\sigma(maxStep) = 50$, $\sigma(currStep) = 0$, $\sigma(steps) = 5$.

The controller *LegC* has the I/O-signature $\Sigma_{LegC} = (\mathcal{V}_{LegC}, \mathcal{O}_{LegC})$, where $\mathcal{V}_{LegC}^{prov} = \{distance\}$, $\mathcal{V}_{LegC}^{req} = \{maxStep, currStep\}$, and $\mathcal{V}_{LegC}^{int} = \{gone\}$. The provided state variable *distance* stores the total distance to be walked by its (connected) leg, the internal state variable *gone* models the distance that has already been moved by its leg. The meaning of the required state variables *maxStep* and *currStep* has already been explained above. The provided operations of *LegC* include *update(a)*. In the following, for the specification of the behavior of *LegC*, we will only consider transitions labeled with $swing(a) \in \mathcal{O}_{LegC}^{req}$ (which is a provided operation of the *Leg* component, hence shared with *LegC*). ■

We are now able to define which labels can occur in a modal I/O-transition system with data constraints. Given an I/O-signature $\Sigma = (\mathcal{V}, \mathcal{O})$, the set $\mathcal{L}(\Sigma)$ of Σ -labels consists of the following expressions:

1. $[\varphi]?m[\pi]$ with $m \in \mathcal{O}^{prov}$ a provided operation, $\varphi \in \mathcal{S}(\mathcal{V}^{prov}, par(m))$ the precondition, $\pi \in \mathcal{T}(\mathcal{V}^{prov} \cup \mathcal{V}^{int}, par(m))$ the postcondition,
2. $[\varphi]!n[\pi]$ with $n \in \mathcal{O}^{req}$ a required operation, $\varphi \in \mathcal{S}(\mathcal{V}^{prov} \cup \mathcal{V}^{req} \cup \mathcal{V}^{int}, par(n))$ the precondition, $\pi \in \mathcal{T}(\mathcal{V}^{req}, par(n))$ the postcondition,
3. $[\varphi]i[\pi]$ with $i \in \mathcal{O}^{int}$ an internal operation, $\varphi \in \mathcal{S}(\mathcal{V}^{prov} \cup \mathcal{V}^{req} \cup \mathcal{V}^{int}, par(i))$ the precondition, $\pi \in \mathcal{T}(\mathcal{V}^{prov} \cup \mathcal{V}^{int}, par(i))$ the postcondition.

There are three kinds of labels concerning reception (?), sending (!) and internal execution of an operation. In each case, labels are equipped with pre- and post-conditions represented by state and transition predicates over particular sets of state variables.

Input labels. A label $[\varphi]?m[\pi]$ models that if a provided operation m is invoked under the precondition φ then the postcondition π will hold after the execution of m . In this case φ expresses the *assumption* (of the specified component) that the environment will only call m when the component's data state fulfills φ . Hence φ must be a state predicate over the *provided* state variables of the component (possibly containing the operation's formal parameters as logical variables). In particular, no internal state variables are allowed in φ , since it must be possible for the environment to check whether the precondition φ is actually valid upon operation call. The postcondition π models the change of the component's data state caused by the execution of the operation m and hence it is a transition predicate over the *provided* and the *internal* state variables of the component (again possibly containing the operation's formal parameters as logical variables). In this case π expresses the *guarantee* (of the specified component) that after the execution of m the postcondition π holds (if the assumption φ was met)¹. The assumptions and guarantees of a component concerning input labels are part of the contract principle behind data constraints as shown in Table 1.

Output labels. A label $[\varphi]!n[\pi]$ models that a component issues a call to a required operation n under the precondition φ and with postcondition π . Here φ describes the condition under which the operation call is performed by a component. Since the component has access to its own, provided and internal, state variables and can also query its required state variables, the condition φ can contain all kinds of state variables (together with the operation's formal parameters as logical variables). From the contract point of view the component *guarantees* to issue the call to the operation n only if φ holds². The postcondition π of an output label formulates the expectations on the change of the data state performed by the environment after the invoked operation has been executed. Hence π must be a transition predicate over the *required* state variables which expresses an *assumption* on the environment. The assumptions and guarantees of a component concerning output labels accomplish the contract principle behind data constraints as shown in Table 1.

Internal labels. Finally, a label $[\varphi]i[\pi]$ stands for the execution of an internal operation i . In this case φ describes the condition under which the internal operation is executed which, like the precondition in output labels, can again depend on all kinds of state variables. The postcondition π models the change of the component's data state caused by the execution of the internal operation i

¹ The environment can in fact only see the observable consequences of the guarantee π w.r.t. the provided state variables.

² In fact, the environment can only see the observable consequences of the guarantee φ w.r.t. the required state variables.

Table 1. Data constraints as contracts

Component
$[\varphi]?m[\pi]$ assume φ guarantee π
$[\varphi]!n[\pi]$ guarantee φ assume π

and hence, like the postcondition in input labels, must be a transition predicate over the provided and the internal state variables.

Example 3. An example of an input label is the Σ_{Leg} -label

$$[a \leq \mathit{maxStep}] ?\mathit{swing}(a) [\mathit{currStep}' = a \wedge \mathit{steps}' = \mathit{steps} + 1]$$

where $\mathit{maxStep}$ and $\mathit{currStep}$ are provided state variables, and steps is an internal state variable of component Leg . The primed expression $\mathit{currStep}'$ in the postcondition indicates that we refer to the value of $\mathit{currStep}$ in the poststate. For component $LegC$, the expression

$$[a = \min(\mathit{distance} - \mathit{gone}, \mathit{maxStep})] !\mathit{swing}(a) [\mathit{currStep}' \leq a]$$

is a valid Σ_{LegC} -label. This label expresses that the output $\mathit{swing}(a)$ happens if the precondition $\varphi \equiv [a = \min(\mathit{distance} - \mathit{gone}, \mathit{maxStep})]$ is satisfied. Note that φ involves all three kinds of state variables: the provided state variable $\mathit{distance}$, the required state variable $\mathit{maxStep}$ and the internal state variable gone . The postcondition $\pi \equiv [\mathit{currStep}' \leq a]$ involves (beside the parameter a) the required state variable $\mathit{currStep}$. ■

Definition 2 (MIOD). A modal I/O-transition system with data constraints

$$S = (\mathit{states}_S, \mathit{start}_S, \Sigma_S, \Delta_S^{\mathit{may}}, \Delta_S^{\mathit{must}})$$

consists of a set of states states_S , the initial state $\mathit{start}_S \in \mathit{states}_S$, an I/O-signature Σ_S , a may-transition relation $\Delta_S^{\mathit{may}} \subseteq \mathit{states}_S \times \mathcal{L}(\Sigma_S) \times \mathit{states}_S$, and a must-transition relation $\Delta_S^{\mathit{must}} \subseteq \Delta_S^{\mathit{may}}$. The class of all MIODs is denoted by MIOD . The set of the (syntactically) reachable states of a MIOD S is defined by $\mathcal{R}(S) = \bigcup_{n=0}^{\infty} \mathcal{R}_n(S)$ where $\mathcal{R}_0(S) = \{\mathit{start}_S\}$ and $\mathcal{R}_{n+1}(S) = \{s' \mid \exists (s, \ell, s') \in \Delta_S^{\mathit{may}} \text{ such that } s \in \mathcal{R}_n(S)\}$.

Example 4. The MIOD S_{Leg} specifying the behavior of the Leg component is shown in Fig. [1](#). Must-transitions are drawn with solid arrows and may-transitions with dashed arrows. Preconditions are written above or to the left of operations;

postconditions are written below or to the right of operations. Pre- and postconditions of the form $[true]$ are omitted.

In the initial state 0, *Leg* can receive *init()* which has the effect of initializing the internal state variable *steps* with 0. In state 1 the leg is able to receive *lift()*, and then *swing(a)*; in the latter case, the leg component assumes that the value of the parameter *a* does not exceed the limit *maxStep* more than ten percent; the maximal step size *maxStep* is a provided state variable, hence visible for the connected leg controller. The guarantee of the leg component is then, that the current step size is *a* and that the number of steps is increased by one. Next, in state 3, the leg is put on the ground when the operation *drop()* is received, and then, on reception of the operation call *retract()*, the leg is pulling the body forward. Finally, an update message is sent to inform the leg controller how far the leg has been moved forward. ■

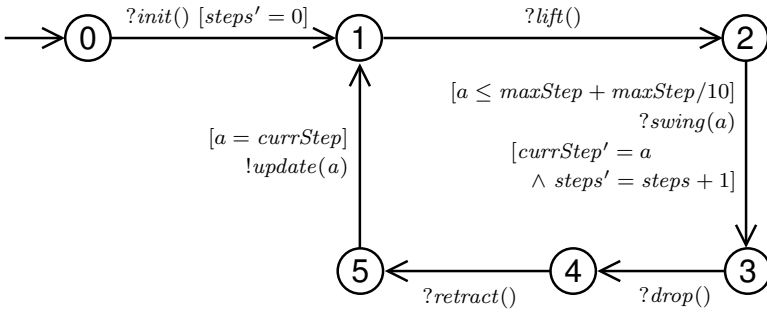


Fig. 1. Specification S_{Leg} of the *Leg* component

3 Refinement of MIODs

The basic idea of *modal* refinement is that any required (*must*) transition in the abstract specification must also be present in the concrete specification. Conversely, any allowed (*may*) transition in the concrete specification must be allowed by the abstract specification. Modal refinement has the following consequences: A concrete specification may omit allowed transitions, but is required to keep all must-transitions. Moreover, it is not allowed to perform more transitions than the abstract specification admits.

Concerning the impact of data constraints, let us first consider required (i.e. *must*) transitions of an abstract MIOD, say T , see Def. 3(1). Obviously, any such transition with a precondition φ_T must also be executable in a refinement S , whenever φ_T is valid. Hence there should be a corresponding must-transition in S whose precondition φ_S does not require more than φ_T does. This condition is independent of the kind of the labels. Concerning postconditions the situation is different because postconditions are not related to the executability of transitions

but rather to the specification of admissible poststates after a transition has fired. In this case, if the transition of T concerns input or internal labels, the corresponding transition of the refinement S should lead to a postcondition π_S which guarantees the postcondition π_T of T (expected, for instance, by a user of a provided operation who relies on the postcondition given in the abstract specification T , see 1(a) in Def. 3). However, if a transition of T concerns an output label, then the postcondition π_T expresses the expectation of T about the next state of the environment. Then, a refinement S should not expect more than the abstract specification does, i.e. π_S should be at most weaker than π_T , see 1(b) in Def. 3.

Let us now consider may-transitions of the concrete MIOD S , see Def. 3(2). Obviously, any such transition with a precondition φ_S must be allowed by the abstract specification T , whenever φ_S is valid. Hence, there should be a corresponding may-transition in T whose precondition φ_T is at most weaker than φ_S and this condition is again independent of the kind of the labels. As explained above the situation is different when considering postconditions since they are not related to the executability of transitions. Therefore, in this case, the kind of the transitions (may or must) is irrelevant and hence the requirements 2(a)(b) coincide with the requirements 1(a)(b) in Def. 3.

In summary, we can observe that the implication direction concerning preconditions in a refinement depends on the kind of the transitions (may or must) while the implication direction concerning postconditions in a refinement depends on the kind of the labels (input, internal, or output).

Definition 3 (Modal Refinement of MIODs). *Let S and T be two MIODs with the same I/O-signature $\Sigma = (\mathcal{V}, \mathcal{O})$. A binary relation $R \subseteq \text{states}_S \times \text{states}_T$ is a modal refinement between the states of S and T iff for all $(s, t) \in R$*

1. *(from abstract to concrete) for all $op \in \mathcal{O}$, if $(t, [\varphi_T]op[\pi_T], t') \in \Delta_T^{\text{must}}$, then there exist $s' \in \text{states}_S$ and a transition $(s, [\varphi_S]op[\pi_S], s') \in \Delta_S^{\text{must}}$ such that $(s', t') \in R$, $\models \varphi_T \Rightarrow \varphi_S$, and the following holds:*
 - (a) *if $op \in \mathcal{O}^{\text{prov}} \cup \mathcal{O}^{\text{int}}$ then $\models \pi_T \Leftarrow \pi_S$,*
 - (b) *if $op \in \mathcal{O}^{\text{req}}$ then $\models \pi_T \Rightarrow \pi_S$.*
2. *(from concrete to abstract) for all $op \in \mathcal{O}$, if $(s, [\varphi_S]op[\pi_S], s') \in \Delta_S^{\text{may}}$, then there exist $t' \in \text{states}_T$ and a transition $(t, [\varphi_T]op[\pi_T], t') \in \Delta_T^{\text{may}}$ such that $(s', t') \in R$, $\models \varphi_T \Leftarrow \varphi_S$, and the following holds:*
 - (a) *if $op \in \mathcal{O}^{\text{prov}} \cup \mathcal{O}^{\text{int}}$ then $\models \pi_T \Leftarrow \pi_S$,*
 - (b) *if $op \in \mathcal{O}^{\text{req}}$ then $\models \pi_T \Rightarrow \pi_S$.*

A state $s \in \text{states}_S$ refines a state $t \in \text{states}_T$, written $s \leq_m t$, iff there exists a modal refinement between the states of S and T containing (s, t) . S is a modal refinement of T , written $S \leq_m T$, iff $\text{start}_S \leq_m \text{start}_T$.

In the following, we will illustrate the concepts of refinement and, later on, compatibility by means of our running example. We will focus here on the treatment of data constraints by considering small excerpts of corresponding MIODs.

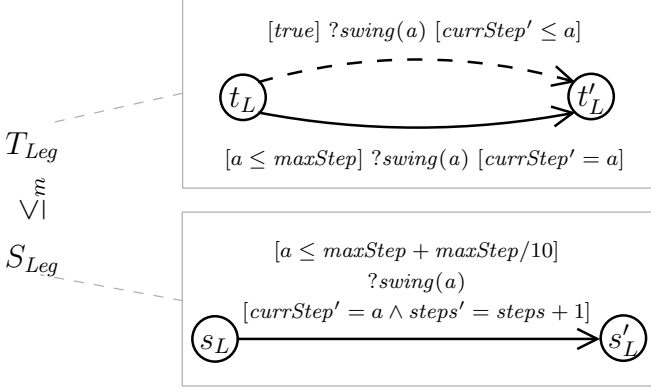


Fig. 2. Refinement S_{Leg} of an abstract specification T_{Leg}

Example 5. Fig. 2 shows excerpts of two MIODs specifying the component Leg , an abstract specification T_{Leg} and the more concrete specification S_{Leg} , see also Fig. 1, which refines T_{Leg} , i.e. $S_{Leg} \leq_m T_{Leg}$. The abstract specification T_{Leg} expresses that under the precondition $[a \leq \maxStep]$ a call to the operation $swing(a)$ must be accepted in any implementation such that in the state after execution of the operation, the postcondition $[currStep' = a]$ is satisfied. The proper may-transition of T_{Leg} says that also such implementations are allowed which accept the operation call $swing(a)$ in states not satisfying $[a \leq \maxStep]$, and then the postcondition $[currStep' \leq a]$ must be satisfied in the next state.

In the refinement S_{Leg} there is only a must-transition. The refinement relation holds since, as required by condition 1 in Def. 3, for the must-transition in T_{Leg} there is a corresponding must-transition in S_{Leg} such that the precondition is weakened (by allowing also values of the parameter a exceeding the maximal step size at most by ten percent) and, according to 1(a) in Def. 3, the postcondition is strengthened by requiring additionally $[steps' = steps + 1]$. Conversely, for the direction from S_{Leg} to T_{Leg} , condition 2 in Def. 3 requires that the must-transition (which is also a may-transition) in S_{Leg} is allowed by the abstract specification T_{Leg} . This is indeed the case because for the transition in S_{Leg} there is a corresponding may-transition in T_{Leg} such that the precondition is weakened in T_{Leg} and, according to 2(a) in Def. 3, the postcondition in S_{Leg} is stronger than the corresponding postcondition in T_{Leg} . ■

4 Composition and Compatibility of MIODs

MIODs can be composed to specify the behavior of concurrent systems of interacting components with data constraints. The composition operator extends the synchronous composition of modal I/O-transition systems [9]. Like for MIOs, we need some syntactic restrictions under which two MIODs are composable. First, we require that overlapping of operations only happens on complementary types and that the same holds for state variables.

Definition 4 (Composability of I/O-Signatures). *Two I/O-signatures $\Sigma_S = (\mathcal{V}_S, \mathcal{O}_S)$ and $\Sigma_T = (\mathcal{V}_T, \mathcal{O}_T)$ are composable if*

1. $\mathcal{V}_S \cap \mathcal{V}_T = (\mathcal{V}_S^{prov} \cap \mathcal{V}_T^{req}) \cup (\mathcal{V}_T^{prov} \cap \mathcal{V}_S^{req})$,
2. $\mathcal{O}_S \cap \mathcal{O}_T = (\mathcal{O}_S^{prov} \cap \mathcal{O}_T^{req}) \cup (\mathcal{O}_T^{prov} \cap \mathcal{O}_S^{req})$.

The set $\mathcal{V}_S \cap \mathcal{V}_T$ of shared state variables will be denoted by $sh(\mathcal{V}_S, \mathcal{V}_T)$ and the set $\mathcal{O}_S \cap \mathcal{O}_T$ of shared operations will be denoted by $sh(\mathcal{O}_S, \mathcal{O}_T)$.

When two composable I/O-signatures are actually composed, the shared state variables become internal. Likewise, the shared operations become internal representing the synchronization of provided and required operations.

Definition 5 (Composition of I/O-Signatures). *The composition of two composable I/O-signatures $\Sigma_S = (\mathcal{V}_S, \mathcal{O}_S)$ and $\Sigma_T = (\mathcal{V}_T, \mathcal{O}_T)$ is the I/O-signature $\Sigma_S \otimes \Sigma_T = (\mathcal{V}_S \otimes \mathcal{V}_T, \mathcal{O}_S \otimes \mathcal{O}_T)$ where*

$$\begin{aligned} (\mathcal{V}_S \otimes \mathcal{V}_T)^{prov} &= (\mathcal{V}_S^{prov} \uplus \mathcal{V}_T^{prov}) \setminus sh(\mathcal{V}_S, \mathcal{V}_T), \\ (\mathcal{V}_S \otimes \mathcal{V}_T)^{req} &= (\mathcal{V}_S^{req} \uplus \mathcal{V}_T^{req}) \setminus sh(\mathcal{V}_S, \mathcal{V}_T), \\ (\mathcal{V}_S \otimes \mathcal{V}_T)^{int} &= \mathcal{V}_S^{int} \uplus \mathcal{V}_T^{int} \uplus sh(\mathcal{V}_S, \mathcal{V}_T), \end{aligned}$$

$$\begin{aligned} (\mathcal{O}_S \otimes \mathcal{O}_T)^{prov} &= (\mathcal{O}_S^{prov} \uplus \mathcal{O}_T^{prov}) \setminus sh(\mathcal{O}_S, \mathcal{O}_T), \\ (\mathcal{O}_S \otimes \mathcal{O}_T)^{req} &= (\mathcal{O}_S^{req} \uplus \mathcal{O}_T^{req}) \setminus sh(\mathcal{O}_S, \mathcal{O}_T), \\ (\mathcal{O}_S \otimes \mathcal{O}_T)^{int} &= \mathcal{O}_S^{int} \uplus \mathcal{O}_T^{int} \uplus sh(\mathcal{O}_S, \mathcal{O}_T). \end{aligned}$$

Two MIODs are composable if their signatures are composable and if the labels occurring on their transitions satisfy certain syntactic constraints. Condition 1 in Def. 6 is technically necessary to ensure that the composition of MIODs defined below is well-formed. For instance, condition 1(a) requires that preconditions of non-shared, provided operations must not include shared state variables; otherwise the precondition of a provided operation in the composition would involve internal state variables which is not allowed. Condition 2 in Def. 6 is needed from an intuitive point of view. For instance, condition 2(a) requires that for shared, provided operations preconditions of one MIOD can only talk about provided variables which are known from the other MIOD as required variables.

Definition 6 (Composability of MIODs). *Two MIODs S and T are composable if their signatures $\Sigma_S = (\mathcal{V}_S, \mathcal{O}_S)$ and $\Sigma_T = (\mathcal{V}_T, \mathcal{O}_T)$ are composable, if for all transitions $(s, [\varphi_S]op[\pi_S], s') \in \Delta_S^{\text{may}}$ we have*

1. if $op \notin sh(\mathcal{O}_S, \mathcal{O}_T)$ then
 - (a) if $op \in \mathcal{O}_S^{prov}$ then $\varphi_S \in \mathcal{S}(\mathcal{V}_S^{prov} \setminus sh(\mathcal{V}_S, \mathcal{V}_T), par(op))$,
 - (b) if $op \in \mathcal{O}_S^{req}$ then $\pi_S \in \mathcal{T}(\mathcal{V}_S^{req} \setminus sh(\mathcal{V}_S, \mathcal{V}_T), par(op))$,
2. if $op \in sh(\mathcal{O}_S, \mathcal{O}_T)$ then
 - (a) if $op \in \mathcal{O}_S^{prov}$ then $\varphi_S \in \mathcal{S}(\mathcal{V}_S^{prov} \cap \mathcal{V}_T^{req}, par(op))$,
 - (b) if $op \in \mathcal{O}_S^{req}$ then $\pi_S \in \mathcal{T}(\mathcal{V}_S^{req} \cap \mathcal{V}_T^{prov}, par(op))$,

and if the same holds symmetrically for all transitions $(t, [\varphi_T]op[\pi_T], t') \in \Delta_T^{\text{may}}$.

The composition of two MIODs S and T synchronizes transitions whose labels refer to shared operations. For instance, a transition with label $[\varphi_S]?op[\pi_S]$ of S is synchronized with a transition with label $[\varphi_T]!op[\pi_T]$ of T which results in a transition with label $[\varphi_S \wedge \varphi_T]op[\pi_S \wedge \pi_T]$ where the original preconditions (postconditions resp.) are combined by logical conjunction. Transitions whose labels concern shared operations which cannot be synchronized are dropped while all other transitions are interleaved in the composition. Concerning modalities we follow the MIO composition operator which yields a must-transition if two must-transitions are synchronized and a may-transition otherwise.

Definition 7 (Composition of MIODs). *The composition of two composable MIODs S and T is the MIOD*

$$S \otimes T = (states_S \times states_T, (start_S, start_T), \Sigma_S \otimes \Sigma_T, \Delta_{S \otimes T}^{\text{may}}, \Delta_{S \otimes T}^{\text{must}})$$

where the transition relations $\Delta_{S \otimes T}^{\text{may}}$ and $\Delta_{S \otimes T}^{\text{must}}$ are defined by the following rules:

$$\frac{(s, [\varphi_S]op[\pi_S], s') \in \Delta_S^\gamma, (t, [\varphi_T]op[\pi_T], t') \in \Delta_T^\gamma}{((s, t), [\varphi_S \wedge \varphi_T]op[\pi_S \wedge \pi_T], (s', t')) \in \Delta_{S \otimes T}^\gamma} \quad \begin{array}{l} op \in sh(\mathcal{O}_S, \mathcal{O}_T), \\ \gamma \in \{\text{may}, \text{must}\} \end{array}$$

$$\frac{(s, [\varphi_S]op[\pi_S], s') \in \Delta_S^\gamma, t \in states_T}{((s, t), [\varphi_S]op[\pi_S], (s', t)) \in \Delta_{S \otimes T}^\gamma} \quad op \notin sh(\mathcal{O}_S, \mathcal{O}_T), \gamma \in \{\text{may}, \text{must}\}$$

$$\frac{(t, [\varphi_T]op[\pi_T], t') \in \Delta_T^\gamma, s \in states_S}{((s, t), [\varphi_T]op[\pi_T], (s, t')) \in \Delta_{S \otimes T}^\gamma} \quad op \notin sh(\mathcal{O}_S, \mathcal{O}_T), \gamma \in \{\text{may}, \text{must}\}$$

Example 6. Assume given the MIOD S_{Leg} of Fig. 1 and a MIOD S_{LegC} describing the behavior of the controller component $LegC$. Fig. 3 shows the result of the synchronization of the two transitions of S_{LegC} and S_{Leg} concerning the shared

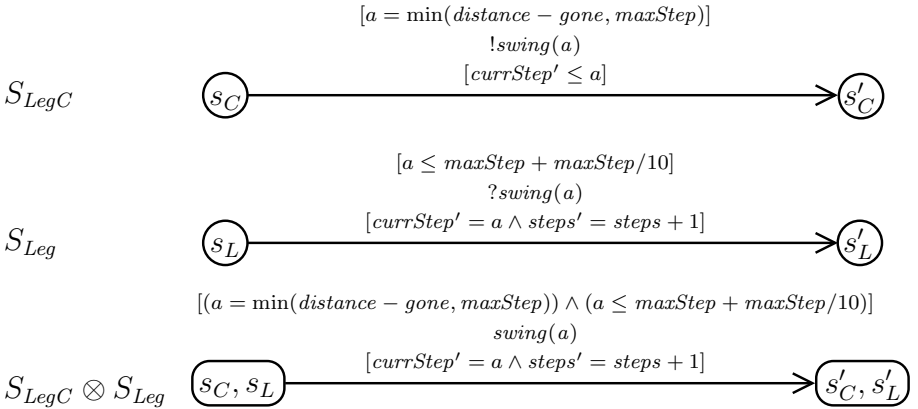


Fig. 3. MIOD composition

operation $swing(a)$ which is an internal operation in the composition $S_{LegC} \otimes S_{Leg}$. Similarly the shared state variable $maxStep$ is an internal state variable in $S_{LegC} \otimes S_{Leg}$. ■

As discussed above, if we want to compose two MIODs it is first necessary to check composability which is a purely syntactic condition. But then it is of course important that the two components work properly together, i.e. are behaviorally compatible. The following compatibility notion builds upon (strong) modal compatibility of MIOs as defined in [4]. From the control point of view (strong) compatibility requires that in any reachable state of the product $S \otimes T$ of two MIODs S and T , if one MIOD *may* issue an output (in its current control state) then the other MIOD is in a control state where it *must* be able to take the corresponding input³. In the context of data states we have the additional requirement that the data constraints of the two MIODs S and T must be compatible. Since the data constraints imposed by a MIOD can be considered as a contract (see Table I) the two contracts according to S and T must match. More precisely, matching means that the mutual assumptions and guarantees of the two MIODs imply each other for any shared operation as illustrated in Table 2. Thus, by combining the control flow and the data state aspects of compatibility we obtain the following compatibility notion for MIODs.

Table 2. Data compatibility of MIODs

S	T	$S \sim T$
$[\varphi_S^m]?m[\pi_S^m]$ assume φ_S^m guarantee π_S^m	$[\varphi_T^m]!m[\pi_T^m]$ guarantee φ_T^m assume π_T^m	$\varphi_S^m \leftarrow \varphi_T^m$ $\pi_S^m \Rightarrow \pi_T^m$
$[\varphi_S^n]!n[\pi_S^n]$ guarantee φ_S^n assume π_S^n	$[\varphi_T^n]?m[\pi_T^n]$ assume φ_T^n guarantee π_T^n	$\varphi_S^n \Rightarrow \varphi_T^n$ $\pi_S^n \leftarrow \pi_T^n$

Definition 8 (Compatibility of MIODs). Let S and T be two composable MIODs. S and T are compatible, denoted by $S \sim T$, if for all reachable states $(s, t) \in \mathcal{R}(S \otimes T)$,

1. if $(s, [\varphi_S]op[\pi_S], s') \in \Delta_S^{\text{may}}$ and $op \in (\mathcal{O}_S^{\text{req}} \cap \mathcal{O}_T^{\text{prov}})$, then there exists a *must*-transition $(t, [\varphi_T]op[\pi_T], t') \in \Delta_T^{\text{must}}$ such that $\models \varphi_S \Rightarrow \varphi_T$, and for all *may*-transitions $(t, [\varphi_T]op[\pi_T], t') \in \Delta_T^{\text{may}}$ it holds that $\models \pi_S \leftarrow \pi_T$;
2. if $(t, [\varphi_T]op[\pi_T], t') \in \Delta_T^{\text{may}}$ and $op \in (\mathcal{O}_T^{\text{req}} \cap \mathcal{O}_S^{\text{prov}})$, then there exists a *must*-transition $(s, [\varphi_S]op[\pi_S], s') \in \Delta_S^{\text{must}}$ such that $\models \varphi_T \Rightarrow \varphi_S$, and for all *may*-transitions $(s, [\varphi_S]op[\pi_S], s') \in \Delta_S^{\text{may}}$ it holds that $\models \pi_T \leftarrow \pi_S$.

³ This concept follows a “pessimistic” approach where two components should be compatible in any environment, in contrast to the “optimistic” approach pursued in [6,9] which relies on the existence of a “helpful” environment.

Example 7. In $S_{LegC} \otimes S_{Leg}$, the state (s_C, s_L) is a reachable state, see Fig. 3. Compatibility holds because for the operation call $swing(a)$ issued by S_{LegC} in state s_C there exists a single must-transition of S_{Leg} which guarantees the reception of $swing(a)$ in state s_L such that the following holds (for any usual satisfaction relation):

$$\begin{aligned} &\models (a = \min(\text{distance} - \text{gone}, \text{maxStep})) \Rightarrow (a \leq \text{maxStep} + \text{maxStep}/10), \\ &\models (\text{currStep}' = a \wedge \text{steps}' = \text{steps} + 1) \Rightarrow (\text{currStep}' \leq a). \quad \blacksquare \end{aligned}$$

We are now able to state our central result which says that compatibility is preserved by refinement and that refinement is compositional (for compatible MIODs). Thus our framework supports independent implementability.

Theorem 1 (Independent Implementability). *Let $S, S', T,$ and T' be MIODs, and assume that S, T as well as S', T' are composable. If $S \sim T, S' \leq_m S$ and $T' \leq_m T$, then $S' \sim T'$ and $S' \otimes T' \leq_m S \otimes T$.*

Proof. We first prove preservation of compatibility. Let $(s', t') \in \mathcal{R}(S' \otimes T')$ be a reachable state in $S' \otimes T'$. It follows from condition 2 of Def. 3 that there exists a reachable state $(s, t) \in \mathcal{R}(S \otimes T)$ such that $t' \leq_m t$ and $s' \leq_m s$. Assume that there exists a transition $(s', [\varphi_{S'}]op[\pi_{S'}], \hat{s}') \in \Delta_{S'}^{\text{may}}$ such that $op \in \mathcal{O}_{S'}^{\text{req}} \cap \mathcal{O}_{T'}^{\text{prov}}$. From $s' \leq_m s$ it follows that there exists $(s, [\varphi_S]op[\pi_S], \hat{s}) \in \Delta_S^{\text{may}}$ such that $\models \varphi_S \Leftarrow \varphi_{S'}$ and $\models \pi_S \Rightarrow \pi_{S'}$. From $S \sim T$ it follows that there exists $(t, [\varphi_T]op[\pi_T], \hat{t}) \in \Delta_T^{\text{must}}$ such that $\models \varphi_S \Rightarrow \varphi_T$. Since $t' \leq_m t$ we can conclude that there exists $(t', [\varphi_{T'}]op[\pi_{T'}], \hat{t}') \in \Delta_{T'}^{\text{must}}$ such that $\models \varphi_T \Rightarrow \varphi_{T'}$. It follows that $\models \varphi_{S'} \Rightarrow \varphi_{T'}$. Then, we must additionally show that for all accepting may-transitions of T' , the postcondition matches $\pi_{S'}$ which is again straightforward.

For the proof of compositionality of modal refinement we define $\leq'_m \subseteq (\text{states}_{S'} \times \text{states}_{T'}) \times (\text{states}_S \times \text{states}_T)$ by

$$(s', t') \leq'_m (s, t) \text{ iff } s' \leq_m s, t' \leq_m t, (s', t') \in \mathcal{R}(S' \otimes T') \text{ and } (s, t) \in \mathcal{R}(S \otimes T).$$

Obviously, $(\text{start}_{S'}, \text{start}_{T'}) \leq'_m (\text{start}_S, \text{start}_T)$ and it remains to show that \leq'_m is a modal refinement between the states of $S' \otimes T'$ and the states of $S \otimes T$. The detailed proof can be found in [2].

Example 8. In Fig. 4 the principle of independent implementability is illustrated in terms of our running example showing small excerpts of four MIODs. Starting from the abstract MIODs T_{LegC} and T_{Leg} we first check their compatibility, i.e. $T_{LegC} \sim T_{Leg}$. Then we refine T_{LegC} and T_{Leg} independently of each other by the MIODs S_{LegC} and S_{Leg} , respectively. Thm. 1 guarantees, first, that S_{LegC} and S_{Leg} are compatible (as explained explicitly in Ex. 7) and secondly, that $S_{LegC} \otimes S_{Leg} \leq_m T_{LegC} \otimes T_{Leg}$ holds. \blacksquare

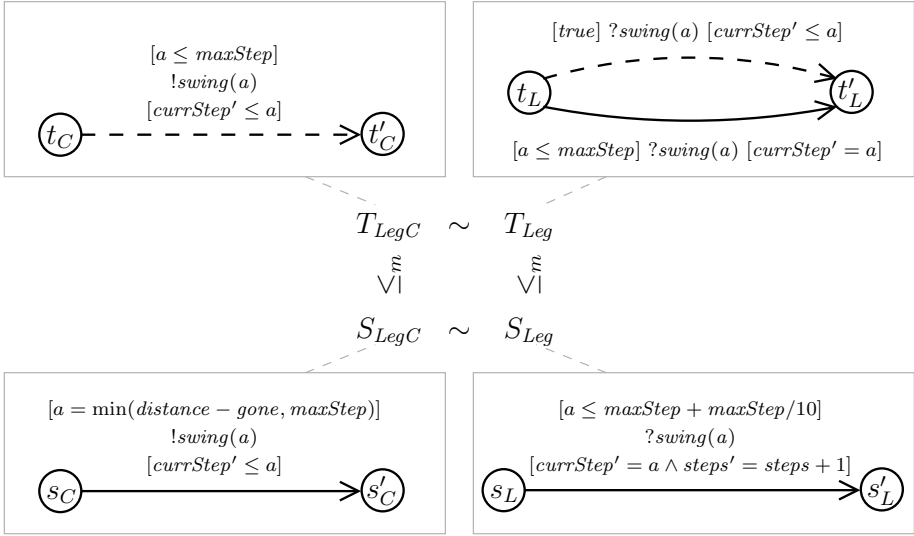


Fig. 4. Compositional refinement of compatible components *LegC* and *Leg*

5 Conclusion

Modal I/O-transition systems (MIOs) provide a flexible framework for the specification and refinement of interacting, concurrent components. In this work we have extended MIOs to take into account data constraints which allow to specify the behavior of components with regard to changing data states. We have shown that our proposal satisfies the requirements of an interface theory guaranteeing preservation of compatibility and compositionality of refinements when moving from abstract to more concrete specifications of interacting components.

We have tried to keep the formalism for refinement as simple as possible at the cost of some restrictions. For instance, one would like to be able to refine a single (abstract) must-transition with some precondition φ by several (concrete) transitions which distribute the precondition over several cases φ_i for $i = 1, \dots, n$, such that $\varphi \Rightarrow \bigvee_i \varphi_i$. Similarly, one could relax the requirements for may-transitions. Moreover, it would not be a problem to integrate state invariants in our framework.

In this paper we have worked on the level of specifications and their refinement and compatibility which are clearly dependent on the syntactical representation of the specification. We have not yet provided a formal semantics which would allow us to define semantic notions of refinement and compatibility. A formal semantics could be delivered in terms of the class of all correct implementations of a specification as done for MIOs in [10] and for behavior protocols without modalities but with data states in [3]. An implementation would then be modeled by a transition system with must-transitions only and with concrete data states given by valuations of the state variables. Concerning the interpretation of

specification transitions with pre- and postconditions, the implementation must guarantee that single specification transitions of a component are executed in an atomic way.

Verification techniques for refinement and compatibility are clearly an important issue for future work. Further next steps are, from the theoretical point of view, to extend our framework by taking into account weak versions of refinement and compatibility abstracting away not only internal actions, as done for MIOs in [10,4], but also internal state variables. From the practical point of view we plan to integrate data constraints in our tool, the MIO Workbench [4], for modal refinement and compatibility checking.

References

1. Barros, T., Ameur-Boulifa, R., Cansado, A., Henrio, L., Madelaine, E.: Behavioural models for distributed Fractal components. *Annales des Télécommunications* 64(1-2), 25–43 (2009)
2. Bauer, S.S., Hennicker, R., Bidoit, M.: A modal interface theory with data constraints. Technical Report 1005, Ludwig-Maximilians-Universität München, Germany (2010)
3. Bauer, S.S., Hennicker, R., Janisch, S.: Behaviour protocols for interacting stateful components. *Electr. Notes Th. Comp. Sci.* 263, 47–66 (2010)
4. Bauer, S.S., Mayer, P., Schroeder, A., Hennicker, R.: On weak modal compatibility, refinement, and the MIO Workbench. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 175–189. Springer, Heidelberg (2010)
5. Bidoit, M., Hennicker, R., Knapp, A., Baumeister, H.: Glass-box and black-box views on object-oriented specifications. In: Proc. SEFM 2004, Beijing, China, pp. 208–217. IEEE Comp. Society Press, Los Alamitos (2004)
6. de Alfaro, L., Henzinger, T.A.: Interface Theories for Component-Based Design. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 148–165. Springer, Heidelberg (2001)
7. Fernandes, F., Royer, J.-C.: The STSLib project: Towards a formal component model based on STS. *Electr. Notes Th. Comp. Sci.* 215, 131–149 (2008)
8. Fischer, C.: CSP-OZ: a combination of Object-Z and CSP. In: Proc. FMOODS, Canterbury, UK, pp. 423–438. Chapman and Hall, Boca Raton (1997)
9. Larsen, K.G., Nyman, U., Wasowski, 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. Larsen, K.G., Nyman, U., Wasowski, A.: On Modal Refinement and Consistency. In: Caires, L., Li, L. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 105–119. Springer, Heidelberg (2007)
11. Larsen, K.G., Thomsen, B.: A Modal Process Logic. In: 3rd Annual Symp. Logic in Computer Science, LICS 1988, pp. 203–210. IEEE Computer Society, Los Alamitos (1988)
12. Mouelhi, S., Chouali, S., Mountassir, H.: Refinement of interface automata strengthened by action semantics. *Electr. Notes Theor. Comput. Sci.* 253(1), 111–126 (2009)

Synchronizing Model and Program Refactoring

Tiago Massoni¹, Rohit Gheyi¹, and Paulo Borba²

¹ Federal University of Campina Grande

{massoni,rohit}@dsc.ufcg.edu.br

² Federal University of Pernambuco

phmb@cin.ufpe.br

Abstract. Object models provide abstract information about software structure, but their maintenance is difficult after refactoring takes place. In Model-Driven Development (MDD), effective transferral of model refactoring changes to programs is problematic, especially if these programs are subject to developer manipulation. Consequently, code-driven approaches end up being adopted. We formalize a theory of synchronizers, which are sequences of behavior-preserving program transformations. This theory makes use of invariant-based refactoring, the key idea behind synchronizers. We also establish and prove a soundness theorem for synchronizers. By uncovering the formal requirements for correct refactoring synchronization, the proved properties point out issues – regarding consistency, refactoring automation and quality – that recur in several MDD settings that employ object models.

1 Introduction

Refactoring [1,2] improves program structure while preserving behavior. Additional benefits can be obtained from *object model* [3,4] *refactoring*, useful for restructuring software abstractions and invariants, by means of *semantics-preserving* transformations. Synchronization of these transformations to source code is essential [5] in Model-Driven Development (MDD) contexts [6].

However, this is an open problem in the MDD community, especially when both models and programs are manipulated [7]. Automatic generation of artifacts has long been known for their limitations – automation is hard to achieve [8]. In fact, the relationship between object model and object-oriented (OO) program constructs may be complex to deal with, and tools often fail to deal with the desired abstraction gap. As a consequence, many projects abandon models early in the life cycle, adhering to code-driven approaches. Methods and tools for – at least partially – removing human interaction in the process are invaluable to the refactoring practice. Several approaches try to deal with the relationship between model and program transformations [9,10,11,12,13], although, to the best of our knowledge, none has analyzed specific aspects of refactoring and synchronization issues between object models and source code.

This article presents a formal model for synchronized refactoring of object model and programs by means of *proven* primitive semantics-preserving transformations. In particular, our theory is centered on a model-driven approach on

which each model transformation is associated with a *synchronizer*, a sequence of program transformations, which (1) updates code declarations and (2) adapts statements according to the modified declarations – as explained in Section 3. This unidirectional (model to code) approach to synchronization presents a fundamental effect: it allows powerful program refactoring directly from abstract information provided by object model invariants. Previous publications delineate the model-driven approach to refactoring [14], and preliminary conclusions on the use of invariants as basis for automatic refactoring [15]. Those contributions are extended in this paper with a description of synchronizers, soundness proofs and discussion (Sections 3, 4 and 5, respectively).

We establish our theory on previous work in primitive model [16] and program transformations [17,18]. Object models in Alloy [3] express objects, relations and invariants equivalently to the core concepts of UML class diagrams. For programs, we consider a Java-like language [19]. These languages are explained in Section 2. A general soundness theorem is defined and proved for synchronizers (Section 4); in this proof, we ensure that synchronizers are program refinements (preserving behavior) and the refactored program is consistent with the refactored model. Consistency is defined in terms of syntax and semantics; only the syntactic consistency must be adjusted for other languages – the semantic mapping is language independent. The need for the proved properties unveils issues that will recur in several MDD contexts that employ object models, related to automation and refactoring quality (Section 5).

2 Languages

In our theory, we consider object models in Alloy [3], and programs in a Java-like language, developed for reasoning about object-oriented programming [19].

2.1 Model Refactoring

Alloy [3] presents formal type system and semantics for writing object models. An Alloy model contains a sequence of *paragraphs*; a *signature* defines a new type. A signature paragraph introduces a basic type and a collection of *relations*, along with their types and constraints on their values. For instance, an object model for a file system defines signatures for `FSObject` and `Name` – `set` defines unconstrained relations. In Alloy version 3, one signature can extend another, establishing that the extended signature is a subset of its supersignature (`File`). In addition, facts are used to package model invariants. In the following fragment, the formula states that, from all file system objects, only directories may contain other objects. The join operator ‘.’ represents relational dereference, and `FSObject - Dir` expresses all `FSObjects` instances that are not directories (set difference has the conventional meaning).

```
sig Name {}
sig FSObject { name: set Name, contents: set FSObject }
fact { (FSObject-Dir).contents = {} }
sig File extends FSObject {}
```

Regarding model transformations in Alloy, a catalog of primitive transformations was proposed [16]. *Algebraic laws* formalize two primitive transformations; equivalence allows application of the law in both directions. As an example of a law, a new subsignature can be introduced or removed from an existing hierarchy (Law 1). An empty subsignature X can be introduced if declared with a fresh name. After this transformation, the supersignature U becomes abstract (defining no direct instances), as denoted by the invariant denoting the objects in X are the objects in U , except those in S or T ($X = U - S - T$). Similarly, X can be removed if it is not being used and no expression exp of its type exists ($exp \leq U$, but $exp \not\leq S$ and $exp \not\leq T$), where \leq denotes subtyping. Some meta-variables are useful as notation: rs represents relations, while $forms$ represents a set of formulas. Each box represents a template with which actual Alloy declarations can match (ps denotes the paragraphs that are not showed in the template). Additionally, below the templates, provisos that ensure transformation correctness are established. (\leftarrow) defines provisos for application from Left-to-Right (L-R), while (\rightarrow) defines provisos for applying the law from Right-to-Left (R-L).

Alloy Law 1 (*introduce subsignature*)

<pre> ps sig U { rsU } sig S extends U { rsS } sig T extends U { rsT } fact F { forms} </pre>	=	<pre> ps sig U { rsU} sig S extends U{ rsS } sig T extends U{ rsT } sig X extends U{} fact F { forms X = U - S - T } </pre>
---	---	---

provided

- (\rightarrow) (1) ps does not declare any paragraph named X ; (2) there is no signature in ps that extends U ;
- (\leftarrow) X does not appear in ps , rsU , rsS , rsT and $forms$; (2) there is no expression exp , where $exp \leq U$ and $exp \not\leq S$ and $exp \not\leq T$, in ps or $forms$.

The equivalence respects the notion proposed for Alloy [20]. The catalog of Alloy laws has been proven sound and complete in a theorem prover [21]. Furthermore, these laws can be used as basis for several applications that require semantics-preserving transformations, such as *model refactorings*. Since primitive laws are simpler – dealing with a few language constructs – they can be more easily proven sound. By construction, a composition of laws is also correct, providing safe refactorings for object models.

2.2 Program Refactoring

The Java-like language is inspired by Banerjee and Naumann’s work [19]; it does not consider interfaces, multithreading or library classes. A program is a set of classes, a *class table* CT , which always includes a class named `Main`, with a main method as the starting point of execution. A generic class declaration is defined

as follows: `class C extends D { \bar{T} \bar{f} ; \bar{M} } .`, where \bar{T} \bar{f} stands for typed fields in the class, while \bar{M} represents a list of methods.

Only **bool** and **unit** (the empty type) are predefined as primitive types in the language; from these types, other primitives may be built. Furthermore, expressions do not have side effects; object construction occurs only as a command. Similarly, method calls occur in special assignments `x := e.m(\bar{e})` defining both side effect and a return value. Methods can be defined recursively, so loops are omitted.

We adapted to this language a complete set of laws of programming [18]. The following law, for instance, establishes that instantiations of a class B can be replaced by instantiations of its superclass A , as long as B is an empty class. This replacement can occur either in the body of A 's methods or any method in the set of class declarations CT (a class table). $CT[exp'/exp]$ denotes a substitution. Replacement is applicable when expressions of type A are not cast with B and B instances are assigned only to A -typed variables. The opposite application is constrained by a proviso: tests involving A -typed variables with B may not work if A 's instances become B instances. fds and mts represent, respectively, fields and methods. A primed metavariable, like mts' , is a transformed version of the unprimed one, as defined in the **where** clause. A sample of other laws used in this paper are presented in a technical report [22].

Law 1 \langle new superclass \rangle

$$\boxed{
 \begin{array}{l}
 \text{class } A \text{ extends } C \{ \\
 \quad fds \\
 \quad mts \\
 \} \\
 \text{class } B \text{ extends } A \{ \} \\
 CT
 \end{array}
 } =
 \boxed{
 \begin{array}{l}
 \text{class } A \text{ extends } C \{ \\
 \quad fds \\
 \quad mts' \\
 \} \\
 \text{class } B \text{ extends } A \{ \} \\
 CT'
 \end{array}
 }$$

where

$CT' = CT[\text{new } A/\text{new } B]$

$mts' = mts[\text{new } A/\text{new } B]$

provided

(\rightarrow) (1) B is not used in type casts or tests in CT or mts for expressions of type A ;

(2) $x := \text{new } B$ only appears if $\text{type}(x) \leq A$;

(\leftarrow) Variables of type $T \leq A$ are not involved in tests with type B .

In addition to equivalence laws, there are laws for *class refinement* that involve internal representation changes such as addition and removal of private fields. In these laws, simulation is established within a class by a constructor making the coupling invariant true and every method executing on a valid state and resulting in another valid state, as they are based on Morgan's refinement notion [23]. These laws have been proven sound and complete as well, although within a language lacking object references [18]. In order to avoid this limitation in work, we guarantee modular reasoning by using *confinement* as a requirement for programs.

Ownership confinement [24] is a discipline for controlling aliasing in object-oriented languages, restricting access to designated *representation objects* (reps), except through their *owners*, to avoid representation exposure [19]. An owner is a class that maintains representation objects stored in the fields of its

objects. Banerjee and Naumann [19] present a number of static analysis rules for ensuring a property called by them *safety*, which is shown in their work to imply confinement. The input is a class table and its division into three sets of classes: *Own* and *Rep*, defining the possibly non-disjoint sets of owner and representation classes, respectively, and *Client* with all other classes. The analysis is modular, as only *Own* and *Rep* code is constrained (with one exception, for **new** commands). They present the rules as follows:

1. Public methods declared in *Own* or subclasses cannot return *Rep* types; otherwise, references to internal objects might leak to clients;
2. Methods inherited or declared by *Own* cannot have parameters of *Rep* types; otherwise, non-owner subclasses might have access to *Rep* instances;
3. *Rep* classes cannot inherit any methods from non-*Rep* superclasses; for instance, a method could return **self** to a client, which is highly undesirable;
4. For any field access $e.f$, if e is of type *Own*, it cannot access fields of type *Rep*, unless e is **self**; this rule must be checked only for public fields, as it is guaranteed by type safety for private fields;
5. For assignments $x := \text{new } B$ in *Client*, B cannot be *Rep* or any of its subclasses; otherwise, these clients would have direct access to *Rep* instances;
6. For method calls $x := e.m(\bar{e})$: (1) if e is a *Client* object, and the call is within *Own* or *Rep* (or subclasses), m cannot have *Rep* parameters (otherwise *Rep* instances could leak); also, (2) if the call is within *Own*, m is declared in *Own*, and e is **self**, parameters and return may be *Rep* type. The second case in fact weakens the confinement constraints with a condition that can be detected by static analysis.

3 Synchronization

Given a specific consistency relationship between object models and programs, we formalize a unidirectional approach of model-driven refactoring by applying *synchronizers*.

3.1 Synchronizers

For object models, we adopt the approach of primitive transformations being composed into refactorings. To each Alloy law from the catalog we associate a *synchronizer* – set of conditional program transformations disciplined by laws of programming – to be applied to a program, making it consistent with the transformed object model. The mechanics of the synchronization is depicted in Figure 1, where OM represents an object model, and P a program. The first step is the application of a model refactoring (a) – in this case, made up of two Alloy laws, X and Y, applied from L-R. Next, each applied law is associated with a synchronizer (depicted as “corresponds” in Figure 1; for instance, Law X corresponds to **Sync X**), applied to P. The sequential application of the synchronizers in (b) results in a synchronized program.

A synchronizer carries out program refactoring by applying a sequence of law applications, on the assumption of the consistency relationship defined in the next section. The only preconditions for the application of synchronizers is that the program must have confinement for a subset *Own* of classes and in syntactic and semantic consistency with the previous version of the model. Therefore, the synchronizers are especially conceived to exploit the model invariants that are known to be met by the program; program transformations are specialized with

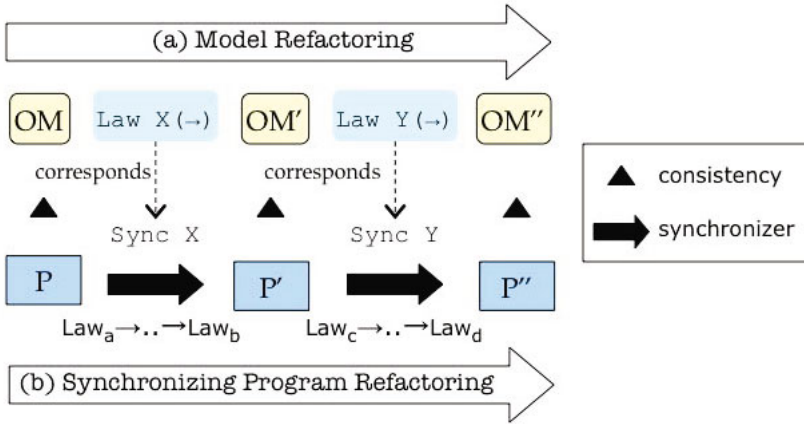


Fig. 1. Model-driven refactoring with synchronizers

high-level assumptions about the program. In fact, we analyzed each Alloy law and conceived synchronizers for both application directions.

A synchronizer must then exhibit the following characteristics: it rewrites programs for updating corresponding abstractions that were refactored in the object model, and preserves program behavior. Therefore, every synchronizer fulfills a few requirements: its application results in programs that refine the previous version and establish consistency with the refactored model, syntactically and semantically, as explained next. In addition, confinement is maintained.

3.2 Consistency Relationship

A desirable property of object models is abstraction; ideally, they can be implemented by several structurally-distinct programs, as long as the invariants hold during their executions. Structures in the model must be somehow implemented in the program, offering a basis for evaluating whether the modeled constraints are met. We call this correspondence *syntactic consistency*. Given a syntactic consistency relationship, fulfillment of model invariants by the executions of a given program is regarded as *semantic consistency*.

We chose a particular syntactic consistency: there must be one direct class for each signature declared in the model (for simplicity, this specification relies on the equality between names, although a mapping between names could be easily established as well). Also, all supersignatures of a signature must have corresponding superclasses of class S , indicating that more superclasses may be declared in the program, but the modeled hierarchy is maintained. Likewise, every relation is mapped to one field with an exactly matching type, with one additional constraint: relations with single multiplicity (yielding a scalar value) are mapped to single field, whereas relations with set multiplicity must be mapped to collection-type fields. Additional classes, fields or methods can be freely declared in the program.

Regarding semantics, an Alloy model defines valid states for a given system – *interpretations* [16] – that contain mappings of signatures and relation names to sets of *object values*. Object values may be single objects for sets and pairs of objects for relations. We consider the semantics of an object model in Alloy as the set of all valid interpretations satisfying all modeled invariants. Each of these interpretations consists of all valid assignments of values to signatures and the relation names. All modeled invariants – implicit or explicit [16] – are satisfied. Invariants are implicit when they constrain the model but are not declared in facts, such as implicit constraints from **extends**.

For programs, states are formalized as *heaps* of object values, mapping class names to sets of objects and field names to pairs of object values (references). If an object in a heap contains a field storing a null value, no pair of values exists with that object as the first member. The semantics of a program is given by *the set of sequences of heaps* resulting from all possible execution traces – depending on the possible program inputs.

It would be straightforward to consider all heaps from every execution trace; however, this approach does not truly reflect the real intentions of consistency checking, since some heaps may be acceptably invalid at some well-defined points of the program. We adopt a specification methodology by Barnett et al. [25], in which every object is added a special *validity field*. If this field has a true value, the invariants over its state should hold, and consistency checking is only performed when all objects are valid. This field can only be modified through the use of two special statements, **unpack** and **pack**. The command **unpack** obj sets the field to false, while **pack** obj does the opposite.

Our semantic consistency regards solely valid heaps (which we call *heaps of interest*). A program is in semantic consistency with a model if, and only if, it is in syntactic consistency, and, for every valid heap from its execution, there is a corresponding interpretation from the semantics of the model.

3.3 Examples of Synchronizers

For each applied Alloy law, two synchronizers are defined. In this paper we show two synchronizers associated with Law [1] introduce and remove subclass. We define synchronizers in a notation for *refinement tactics*, based on the Angel language [26]. Angel constructs are appropriate for describing law applications, with the needed arguments. Tactics may be a simple law applications, with the law name with arguments. A law application may have two possible outcomes: if all provisos are satisfied, then program is transformed. Otherwise, the application of the law fails. For instance, **law** *newSuperclass*(U, X, \rightarrow) applies Law A (*new superclass*) to the program, with three arguments: the superclass (U), the subclass (X) and the application direction (\rightarrow : from L-R). A special atomic tactic, **skip**, always succeeds, leaving the program unchanged.

In order to sequentially composing two tactics, the $t_1; t_2$ construct can be used. Similarly, tactics combined in alternation have the form $(t_1|t_2)$. First, t_1 is

applied to the program; if this application is successful, then the composite tactic succeeds. Otherwise t_2 is applied. Finally, if t_2 fails, then the whole tactic aborts (which is a more critical situation than failure). When the tactic contains many choices, the first choice that succeeds is selected. In addition, the language allows us to define pattern matching within a program, with the constraint **applies to**. For instance, **applies to** $cmd[(X)e]$ **do** t applies the t tactic to every command in the program that includes an expression cast with X .

Introduce Subclass. Law \square (L-R) introduces a subsignature for one of the declared signatures. This makes U abstract according to the modeled invariant ($X=U-S-T$). Here we show the associated synchronizer, that accepts a consistent program. However additional classes can be declared in the program hierarchy.

```
Tactic introduceSubclass( $X, U : Class$ )
  (law rename( $X, X'$ ) | skip);
  law classElimination(setExtends( $X, U$ ),  $\leftarrow$ );
  law newSuperclass( $U, X, \rightarrow$ );
end
```

The trivial law $rename(X, X')$ renames the X class. If it fails (for the case in which the class is not declared), nothing happens (**skip**). If class X is already present, it is freely renamed to X' , because X is considered in this case an *implementation detail* that was not modeled. This action does not have impact on the consistency, as renamed declarations are not in the model. Next, the synchronizer introduces the new class X as a direct subclass of U , with Law *class elimination* [22]. **setExtends** makes X a direct subclass of U . Other subsignatures of U will be declared as classes, although their inheritance relationship with U may be indirect – implementation-only subclasses are allowed. Finally, every U object creation in the entire program is replaced by X instantiations, by Law \square from L-R (**[new X/new U]**).

The synchronizer provides evidence on how model-driven refactoring can improve tool support for refactoring, since the semantic properties from object models can aid refactoring automation. In *introduceSubclass*, the program is refactored to a specific configuration of U objects, making them X instances. This information cannot be obtained solely from the source code, then introducing a plain subclass would not include the changes applied by the synchronizer.

Remove Subclass. The opposite transformation given by Law \square removes subsignature X assuming the invariant ($X=U-S-T$); S and T become the U 's only subsignatures. The synchronizer removes the corresponding X class, although, differently from the model, the program class may declare fields and methods, and may have implementation-only subclasses.

The following definition uses several auxiliary tactics, which are informally shown in this paper; their complete definitions are described in [17].

```
Tactic removeSubclass( $X : Class$ )
  tactic moveUpFields( $X$ );
  tactic moveUpMethods( $X$ );
  tactic changeDeclarationsTypetoSuper( $X$ );
  applies to  $cmd[(X)e]$  do law eliminateCastExpressions( $cmd[(X)e]$ ,  $\rightarrow$ );
```

```

tactic eliminateTypeTests(X, “bool isX() { result := self is X }”);
tactic eliminateNew(X);
law changeSuperFromEmptyToImmediateSuperclass(immedSubs(X),
  super(X), →);
law classElimination(X, →);
end

```

The auxiliary tactic *moveUpFields* pulls up the fields declared in X to the immediate superclass. If the target superclass has any other subclass declaring the moved field, the tactic moves two or more fields with the same name to the superclass in one step; if it is not the case, the single field is moved to the superclass, with Law *move field to superclass* [22]. Next, X 's methods are pulled up as well, with the auxiliary tactic *moveUpMethods*. In this case, the synchronizer must deal with two cases: redefined and non-redefined methods:

- The redefined methods are removed from X and the corresponding method body in the superclass is modified with an **if** command that adds the body of the moved method, using Law *move redefined method to superclass*. Also within the tactic, **super** method calls are eliminated by inlining from **object** to X , top-down in the hierarchy (Law *eliminate super* [22]); for this, all private fields in this hierarchy are first made public;
- The non-redefined methods must be copied to other subclasses of B , with an empty body, so no type errors occur with the new method in B .

After removing its fields and methods, X is replaced by its superclass on declarations over the program, with *changeDeclarationsTypetoSuper*; in this tactic, Law *change field type* and analogous laws are applied. Next, In the main tactic, casts to X are removed with another law (*eliminate casts of expressions*). Consecutively, *eliminateTypeTests* removes type tests involving X , with the following steps:

1. A boolean method **isX** is declared within B and its subclasses. This method is a surrogate for the type tests that are going to be eliminated. The method body returns the value of testing **self** with X and subclasses (in this example, Z);

```

class B { ..
  bool isX() { result := self is X ∨ self is Z }
}

```

2. Every occurrence of **x is X** must be replaced by a special statement, a *parameterized command* [18]. A parameterized command of the form **test := (result := x is X)** is then be replaced by a method call to **isX**. For avoiding null pointer errors, we introduce an **if** statement for ensuring that the expression being tested is not null;

```

if (x=null) then test := false else test := x.isX()

```

3. Additional changes are performed for backing up the **isX** test. Field **type** is introduced and initializations to this field are added to X 's constructor and every constructor in X 's subclasses;


```
class X extends B {   constr { ..; self.type:= "X" } }
class Z extends X {   constr { ..; self.type:= "Z" } }
```

4. Within overriding `isX` implementations, expression `self is X` is replaced with the equivalent expression `self.type = "X"`;

Regarding constructors, `X` declares a constructor that must be replaced, as every `new X` will be rewritten as `new B`. Hereafter, we consider a command of type `x := newX` to be a syntactic sugar for the following sequential composition: `x := new' X; x.newX()`, in which `new'` is the regular instantiation of an object, whose reference is assigned to `x`. It is followed by a call to `newX`, a method of class `X` containing the actual constructor body, used for initializing fields. After defining this replacement for every `X` instantiation, the synchronizer moves `newX` to the superclass `B` (which contains the initialization for the `type` field). After this, the `new' X` commands can be replaced by `new' B` commands in the whole program, with Law A. An excerpt of the result can be seen next.

```
class B { .. string type; ..
  bool isX() { result:= self.type="X" ∨ self is Z }
  unit newX() { { .. self.type:= "X" } } }
class X extends B { } ..
B x:= new' B; x.newX(); ..
```

Finally, the `extends` clause of `X`'s subclasses, then `X` can be eliminated. In general, automated refactorings only remove subclasses when they are not used anywhere in the program. In contrast, this synchronizer can prepare programs when removal of the given subclass is desirable. It replaces all uses of this subclass by the correspondents given by an invariant (stating that class `U` is abstract).

The defined synchronizers follow the correspondence in Table 1. Other laws of modeling do not have corresponding synchronizers, as they deal with syntactic sugar in the model, which does not affect the syntactic consistency.

Table 1. Synchronizers corresponding to Alloy laws

Alloy Law	synchronizer \rightarrow	synchronizer \leftarrow
1.Introduce Relation	<i>introduceField</i>	<i>removeField</i>
2.Introduce Subsignature	<i>introduceSubclass</i>	<i>removeSubclass</i>
3.Introduce Signature	<i>introduceClass</i>	<i>removeClass</i>
4.Introduce Generalization	<i>introduceSuperclass</i>	<i>removeSuperclass</i>
5.Split Relation	<i>splitField</i>	<i>removeIndirectReference</i>
6.Remove Lone Relation	<i>fromOptionalToSetField</i>	<i>fromSetToOptionalField</i>
7.Remove One Relation	<i>fromSingleToSetField</i>	<i>fromSetToSingleField</i>

4 Soundness

A soundness theorem is established for synchronizers. The rationale behind this theorem is the set of conditions for a sound synchronized refactoring. Given these conditions, the compromises for automating the involved transformations can be

analyzed in depth, showing issues that will recur in several MDD contexts. Thus, proofs for synchronizers constitutes the core of our approach. Sound synchronizers depend on the defined consistency relationship and two additional properties: (1) they must express refinements and (2) preserve program confinement.

Theorem 1 is defined for an arbitrary object model (OM), and an arbitrary program (P), in which the application of a law to OM results in OM' , and a synchronizer applied to P results in P' . We define additional predicates from law definitions: $Refines(P', P)$, in which the second argument refines the first, and $Confined(P)$, stating that P satisfies the static analysis confinement rules from Section 2.2, for a subset Own of the class table. $premises(OM, OM', P)$ states the conditions before the application of a synchronizer, as defined in our technical report [22] – an Alloy law applied to OM results in OM' , and consistency and confinement constraints apply to OM and P .

Theorem 1. $\forall OM, OM', P, P' \bullet premises(OM, OM', P) \Rightarrow syntConsistency(OM', P') \wedge Confined(P') \wedge Refines(P', P) \wedge semanticConsistency(OM', P')$

The proof of each synchronizer is split in four *supporting lemmas*. The theorem's meta-variables OM , OM' , P and P' are concretized for each synchronizer.

Lemma 1. $\forall OM, OM', P, P' \bullet premises(OM, OM', P) \Rightarrow syntConsistency(OM', P')$

Lemma 2. $\forall OM, OM', P, P' \bullet premises(OM, OM', P) \Rightarrow Confined(P')$

Lemma 3. $\forall OM, OM', P, P' \bullet premises(OM, OM', P) \Rightarrow Refines(P', P)$

Lemma 4. $\forall OM, OM', P, P' \bullet premises(OM, OM', P) \Rightarrow semanticConsistency(OM', P')$

We proved the synchronizers listed in Table 1, as detailed in [17]; for illustration, here we present the proof for *removeSubclass*. Model and program definitions for the proof are shown in our technical report [22]. Assuming $premises(OM, OM', P)$, we now prove the four previous lemmas.

Proof for Lemma 1. Since no relations are added or removed, the mapping between relations and fields is unchanged. Regarding signatures, X is removed, which is the only class that is removed from the program, establishing the conformance. The hierarchy remains unchanged. Thus $syntConsistency(OM', P')$ follows from the premises.

Proof for Lemma 2. By case analysis on P' for the six static analysis rules of confinement from Section 2.2. For each rule, we justify its maintenance in terms of the premises and P' . In this case, $X \notin Rep$.

1. Class B is the only one to receive new methods. If $B \in Own$, then from $premise(OM, OM', P)$ methods previously in X could never have Rep return types;
2. No inherited methods are added, thus from $premise(OM, OM', P)$ no inherited methods have Rep parameters;
3. Same as above, thus from $premise(OM, OM', P)$, Rep classes do not inherit methods from non- Rep classes;
4. No public fields of Own classes are used outside their declaring module, thus from $premise(OM, OM', P)$ no $e.f$ is seen, unless e is **self**;

5. From premise, if $x := \text{new } X$ was outside *Own* classes, $X \notin \text{Rep}$. Assuming the command is outside *Own*, it is impossible to have $X \notin \text{Rep}$ and $B \in \text{Rep}$, since all subclasses of *Rep* classes are also included. Therefore, property is maintained;
6. From premise, $e.m(\dots)$ within *Own* or *Rep* does not have *Rep* parameters; no changes in parameters or *Rep* are made, so property is maintained.

Proof for Lemma 3. Since the synchronizer steps are exclusively defined as law applications and class refinement [18], P refines P' . Some of the applied laws can be seen in [22].

Proof for Lemma 4. First, any interpretation can be reduced without the mappings to X . This is possible since values of X cannot be interpreted alone, from the given invariant $X = U - S - T$. Consequently, semantics of OM' is the set of interpretations from the semantics of OM , with mapping from X removed. Likewise, for P and P' , the valid heaps of P' are the valid heaps of P reduced in mappings from the X class; this conclusion is implied from the *invariant that is assumed in the program*. Also, the changed commands (type casts and tests) do not add or remove new possible heaps; all X instances that are not B instances are still mapped by U in the heap.

5 Discussion

In this section, the contributions and limitations of our synchronization model are discussed, especially topics related to automation and quality of refactorings.

5.1 Invariants as Basis for Refactoring Automation

The practice of refactoring has been improved by supporting tools, avoiding manual work and increasing trust on semantics preservation. Usually a catalog of refactorings is offered, from which developers can choose the desired transformation for the problem in context. These automated refactorings present *pre-conditions* that are checked against the code subject to refactoring, in order to ensure correctness. While being effective to ensure safe refactorings – at least in theory – it leads to prevention of refactoring on programs that would be eligible if some *semantic assumptions* about the program behavior were considered.

Semantic assumptions about the program can be provided by object models, then synchronizers exploit invariants to increase the applicability of some automated refactorings. Transformations based on these invariants can be applied to programs that would not be eligible for refactoring using the current tools. When removing a subclass, for example, the synchronizer *assumes* the invariant $X = U - S - T$ as true in every reachable program state outside a **unpack/pack**

¹ The laws have been proven semantics preserving for a programming language without references [18]. However, we have proven that these laws are still correct in the presence of confinement [17].

block. In this case, the subclass X can be removed, given that U is an abstract class.

Certainly there are several open questions. For instance, it is not clear how invariants will be automatically identified by a refactoring tool for the application of specific refactorings. Our intuition is that catalogs of program refactorings could be extended with improvements based on invariants, conditionally applied based on a set of invariants.

5.2 Quality of Refactorings

With formal synchronizers, quality factors such as cohesion and legibility still requires some improvement in the resulting program. For instance, the successive application of *removeSubclass* may result in numerous implementations of methods for eliminating type tests, which is clearly amenable to simplification. Therefore, additional refactoring might be necessary, such as inlining these calls and removing methods. These transformations *are also formalizable as laws of programming*. Although theoretically feasible, these law applications could not be automatically applied in the formal model, since our initial assumption is that each synchronizer is recorded and independently applied in order, disregarding the composed refactoring that was applied to the model.

In this scenario, we envisage *developer feedback* as a possible answer to this challenge, in addition to *complementary synchronizers*. In this case, the application of the model refactoring could bring additional information that is then applied in the program refactoring, according to feedback from the developer of a supporting tool. If the developer agrees, a complementary synchronizer, containing the additional law applications, is automatically applied. The outcome of the complementary synchronizer is an improved program, yet still conforming, syntactically and semantically, to the refactored model.

5.3 Consistency and Synchronizers

The required consistency relationship was adjusted during the formalization of synchronizers. Several choices have been considered and this scenario allowed us to gather evidences on how the chosen consistency affects the final results of model-driven program refactorings.

The rule of thumb states that, the more abstract are the models, the looser (different possible implementations for the same object model) is the syntactic consistency relationship. The syntactic mappings between model and program declarations drive the freedom of implementation for modeled signatures and relations. At the end, we adopted a tighter consistency relationship than initially expected: signatures must be implemented as classes and relations as fields in the corresponding class. Nevertheless, the required consistency relationship still preserves some abstraction: methods and additional classes can be freely implemented, and hierarchies can contain more classes than modeled. In addition, the modeled signatures and relations must be implemented in a uniform way, so the synchronization is still compelling for the user. As refactoring is a structural

modification, the declarations in the model must be reflected in the source code for desired transformation; otherwise, the task would be rather pointless.

In addition, the looser is the syntactic consistency, the more complex become the program transformations needed to refactor the program. When giving more freedom of implementation to a specific model declaration, synchronizers must consider every implementation option for this declaration, in order to achieve automation. In this context, synchronizers must be more elaborate, which often *clutters the program, decreasing quality*. This is certainly a trade-off for any synchronization approach.

6 Related Work

The concept of coupled transformation in Lammel’s overview [10] has a close correspondence to our approach. Coupled transformations occur when “two or more artifacts of potentially different types are involved, while transformation at one end necessitates reconciling transformations at other ends such global consistency is reestablished” [10], which is the scenario for model-driven refactoring. This type of synchronization seem to fit into the “symmetric reconciliation” category, in which two distinct transformations – for model and program – are defined for a given consistency relationship, adapting changes according to the specific level of abstraction for which they are defined.

Bidirectional model transformations (bx) [11][12] have the purpose of formalizing synchronization between changed artifacts during the software life cycle (in this approach, model is a comprehensive concept, which includes programs). The proposal includes an abstract definition of synchronizers, which may even be bidirectional (updating both artifacts). Several concepts are similarly formulated – such as unidirectional synchronizers – but no particular approaches of bx are defined for object models and programs. Therefore, our results could be confirmed in such scenario by concretizing bx .

The Harmony tool [13], for instance, is based on the concept of bx . The authors introduce the concept of relational lenses, which are pairs of transformation functions, namely *get* and *putback*, between source and target artifacts. The *get* function transforms a source artifact into a target artifact. Updates can be performed on the target artifacts, then an updated source artifact can be obtained with the *putback* function, with information from the original source artifact and the updated target artifact. Analogously, in our theory *get* is similar to the required consistency relationship, although we avoid generation of artifacts. The source artifact can be a program, and the target can be an object model.

Another related study is carried out by Antiewicz and Czarnecki [27], which formally defines several synchronization alternatives between software artifacts. Their synchronization definitions are applied with the help of formal operators. Several elements are common with our approach, for instance developer feedback for automation and related and independent transformations. Since we focus on a specific type of synchronization (object models to programs), our theory is able to reveal detailed issues about consistency and transformation.

7 Conclusions

In this paper, we formalized a synchronization theory from object model refactoring to object-oriented programs. The theory is backed by a formal infrastructure of primitive transformations proved to be semantics preserving, both for object models and programs, and a specific consistency relationship. Synchronizers are formalized as a sequence of primitive program transformations, explicitly avoiding generation of programs from object models. The investigation unveils several issues concerning consistency, refactoring automation and behavior preservation and quality, providing evidence over the challenges that effective MDD methodologies will face in order to support evolution. Potential improvements for refactoring tools are identified, since the semantic properties from object models can aid refactoring automation. In our synchronizers the invariants expressed in the object model offer semantic information to extend its automatic refactoring capabilities.

The level of abstraction is a key aspect. First, useful model refactoring requires that the main structures be maintained. Second, less restrictions to the source code implementation imply in more transformations required to make the source code conforming to the refactored model, which would lower the quality of the outcome. Assumptions include reliance on the maturity of consistency checking tool support in practice and a closed-world context in which we have access to the full source code of a program.

The theory described in this paper is language specific, although the formalization is amenable to adaptation to other object-oriented languages. In addition, our approach supports only refactoring; dealing with generic evolution in MDD is a challenge for future research. A potential solution might rely on primitive transformations for standard evolution, and model invariants could be used to transform programs accordingly.

Acknowledgment

We'd like to thank Augusto Sampaio, Alexandre Mota, Ana Cristina de Melo, Marcel Oliveira, Juliano Iyoda, and all anonymous reviewers for the relevant comments. This work was partially supported by CNPq grant 477336/2009-4, and the National Institute of Science and Technology for Software Engineering (INES²), also funded by CNPq, grant 573964/2008-4.

References

1. Fowler, M.: Refactoring—Improving the Design of Existing Code. Addison-Wesley, Reading (1999)
2. Opdyke, W.: Refactoring Object-Oriented Frameworks. PhD thesis, UIUC (1992)
3. Jackson, D.: Software Abstractions: Logic, Language and Analysis. MIT Press, Cambridge (2006)

² www.ines.org.br

4. Liskov, B., Guttag, J.: Program Development in Java. Addison-Wesley, Reading (2001)
5. Mens, T., Tourwe, T.: A survey of software refactoring. *IEEE Transactions on Software Engineering* 30, 126–139 (2004)
6. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, Chichester (2004)
7. France, R.B., Rumpe, B.: Model-driven development of complex software: a research roadmap. In: FOSE 2007, pp. 37–54 (2007)
8. Hettel, T., Lawley, M., Raymond, K.: Model synchronisation: Definitions for round-trip engineering. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 31–45. Springer, Heidelberg (2008)
9. Harrison, W., Barton, C., Raghavachari, M.: Mapping UML Designs to Java. In: Proceedings of OOPSLA 2000, pp. 178–187 (2000)
10. Lammel, R.: Coupled software transformations. In: SET, pp. 31–35 (2004)
11. Diskin, Z.: Algebraic models for bidirectional model synchronization. In: Busch, C., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 21–36. Springer, Heidelberg (2008)
12. Stevens, P.: A Landscape of Bidirectional Model Transformations. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 408–424. Springer, Heidelberg (2008)
13. Bohannon, A., Pierce, B., Vaughan, J.: Relational lenses: a language for updatable views. In: PODS, pp. 338–347 (2006)
14. Massoni, T., Gheyi, R., Borba, P.: Formal model-driven program refactoring. In: FASE-ETAPS 2008, pp. 362–376 (2008)
15. Massoni, T., Gheyi, R., Borba, P.: An approach to invariant-based program refactoring. In: Setra Workshop 2006, pp. 91–101 (2006)
16. Gheyi, R., Massoni, T., Borba, P.: A static semantics for alloy and its impact in refactorings. *ENTCS* 184, 209–233 (2007)
17. Massoni, T.: A Model-Driven Approach to Formal Refactoring. PhD thesis, UFPE (2008)
18. Borba, P., Sampaio, A., Cavalcanti, A., Cornélio, M.: Algebraic Reasoning for Object-Oriented Programming. *Science of Computer Programming* 52, 53–100 (2004)
19. Banerjee, A., Naumann, D.A.: Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM* 52, 894–960 (2005)
20. Gheyi, R., Massoni, T., Borba, P.: An abstract equivalence notion for object models. *ENTCS* vol.130, pp.3–21 (2005)
21. Gheyi, R., Massoni, T., Borba, P.: A Complete Set of Object Modeling Laws for Alloy. In: Oliveira, M.V.M., Woodcock, J. (eds.) SBMF 2009. LNCS, vol. 5902, pp. 204–219. Springer, Heidelberg (2009)
22. Massoni, T., Gheyi, R., Borba, P.: Synchronizing model and program refactoring (2010), <http://www.dsc.ufcg.edu.br/~spg/uploads/massoni-tech10.pdf>
23. Morgan, C.: Programming from Specifications, 2nd edn. Prentice-Hall, Englewood Cliffs (1998)
24. Clarke, D.: Object Ownership and Containment. PhD thesis, UNSW (2001)
25. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of Object-Oriented Programs with Invariants. *Journal of Object Technology* 3, 27–56 (2004)
26. Martin, A.: Machine-Assisted Theorem-Proving for Software Engineering. PhD thesis, Penbrooke College (1994)
27. Antkiewicz, M., Czarnecki, K.: Design space of heterogeneous synchronization. In: GTTSE, Braga, Portugal, pp. 3–46 (2008)

A Type-Theoretic Framework for Certified Model Transformations

Daniel Calegari¹, Carlos Luna^{1,2} Nora Szasz², and Álvaro Tasistro²

¹ Instituto de Computación, Universidad de la República, Uruguay
{dcalegar, cluna}@fing.edu.uy

² Facultad de Ingeniería, Universidad ORT Uruguay
{luna, szasz, tasistro}@ort.edu.uy

Abstract. We present a framework based on the Calculus of Inductive Constructions (CIC) and its associated tool the Coq proof assistant to allow certification of model transformations in the context of Model-Driven Engineering (MDE). The approach is based on a semi-automatic translation process from metamodels, models and transformations of the MDE technical space into types, propositions and functions of the CIC technical space. We describe this translation and illustrate its use in a standard case study.

1 Introduction

Model-Driven Engineering (MDE, [1]) is a software engineering paradigm based on the specification of models of a system as the primary development activity. The feasibility of the approach is based on the existence of a semi-automatic construction process driven by model transformations, starting from abstract models of the system and transforming them until an executable model is generated. In consequence, the quality of the whole process strongly depends on the quality of the model transformations. The highest level of quality is achieved by proving desired properties of the transformations. Although formal verification techniques may be expensive, they can be helpful in guaranteeing the correctness of critical applications where no other verification technique is acceptable. Since the MDE approach is intended to succeed in a broad spectrum, we think it is worth exploring how formal verification techniques could be applied within it.

As summarized in Figure 1, a model transformation takes as input a model Ma conforming to a given source metamodel MMa and produces as output another model Mb conforming to a given target metamodel MMb . The model transformation can be defined as well as a model Mt which itself conforms to a model transformation metamodel MMt . There are well known metamodeling languages like the MOF [2] and KM3 [3]. In some cases, there are conditions (called invariants) that cannot be captured by the structural rules of these languages, in which case modeling languages are supplemented with another logical language, e.g. the Object Constraint Language (OCL) [4]. There are different model transformation approaches, as described in [5,6]. In our case we select a

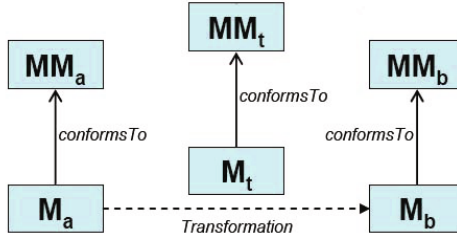


Fig. 1. An overview of model transformation

model-to-model relational approach, which is based on specifying a transformation as a set of relations (rules) that must hold between source and target model elements. Languages within this approach are QVT [7] and ATL [8].

There are basically two levels at which the verification of a model transformation can be exercised: the model and the metamodel levels. Model-level verification works on specific source and target models related by a transformation. Verification techniques within this approach are mainly based on model-checking or simply testing. This approach is in many cases a practical and valuable aid but it cannot ensure the zero-fault level of quality since it checks a finite number of specific cases. Furthermore, there exist well-known limitations such as the state-explosion problem within model checking. An interesting work on the model-level verification of properties is [9] where the language Alloy is used for writing declarative model transformations and the Alloy Analyzer tool is used to conduct fully automated analysis of certain properties with the limitations mentioned above.

In contrast, metamodel-level verification ensures that a model transformation respects certain relations between model instances conforming to the source and target metamodels. This requires the use of formal verification techniques. Works within this approach are [10,11]. The first one is limited to the verification of model refinement transformations whereas the second one is restricted to prove semantic equivalence between source and target models.

We are concerned to *metamodel-level* verification of model transformations, considering *any kind of transformation and properties*. We expect the approach to be helpful whenever *zero-fault model transformations* are required.

We propose the construction of a type-theoretic framework for the certification of model transformations, representing the schema in Figure 1. In particular, we explore the idea of using the Calculus of Inductive Constructions (CIC) [12] as a technical space for dealing with provably correct model transformations. Within this framework, metamodels MM_a and MM_b above are represented as inductive types. On the other hand, each transformation rule of the model transformation M_t is represented as a logical formula (called *TRule*) of $\forall\exists$ form stating that for every model element satisfying a certain (pre-)condition, there exists a target model element which stands in the relation specified by the transformation rule with the source model element.

The correctness of the model transformation is stated as the following logical formula.

$$\forall Ma:MMa. (Pre(Ma) \rightarrow \forall Mb:MMb. (TRules(Ma,Mb) \rightarrow Post(Ma,Mb)))$$

where *Pre* is a translation of the source invariants, *TRules* is the conjunction of the transformation rules, and *Post* is a translation of the target invariants plus any other desired property to be proved. A proof of this formula ensures that the transformation rules satisfy the target invariants as well as the desired properties. We propose a semi-automatic translation process from the MDE technical space –used by developers– to the CIC technical space –used for formal verification.

The choice of the CIC is dictated by its very considerable expressive power as well as by the fact that it is supported by a tool of industrial strength, namely the Coq proof assistant [13]. As one example of its applicability, Coq has been used for the development and formal verification of a compiler of a large subset of the C programming language [14].

The idea of using type theory in the context of MDE has been formulated before by Poernomo in [15,16]. He formulates a type theory of his own –a variant of Martin-Löf’s constructive type theory– and outlines a method for representing MOF models as types. Then, he follows the classical approach in type theory where pre- and post-conditions are represented as types, and a program (transformation) is derived as a function between those types. Our work differs in the representation of metamodels as will be explained later. Another important difference is that he performs program derivation to obtain a transformation whereas we translate a given transformation as a formula and verify this translation with respect to certain pre- and post-conditions. Finally, we base our proposal on an existent type theory with its corresponding supporting verification tool, which allows us to put into practice the ideas presented, unlike the works in [15,16].

As compared to previous work by the authors, the representation of model transformations described here differs substantially from the one presented in [17], as will be explained later.

The remainder of the paper is structured as follows. We first describe our framework in Section 2. In Section 3 we give some details about the formal representation of models and metamodels, and about model transformations in Section 4. Then, in Section 5 we explain how properties are verified. Finally, in Section 6 we present a short summary with concluding remarks and an outline of further work.

2 Outline of the Approach

We use the Calculus of Inductive Constructions (CIC) as a technical space for dealing with provably correct model transformations. In the following sections the CIC is introduced and our framework is outlined, using a running example.

2.1 The CIC as a Technical Space

The CIC is a type theory, i.e. in brief, a higher order logic in which the individuals are classified into a hierarchy of types. The types work very much as in strongly typed functional programming languages which means that, to begin with, there are basic elementary types, recursive types defined by induction like lists and trees (called inductive types) and function types. A (dependent) record type is a non-recursive inductive type with a single constructor and projection functions for each field of the type. An example of inductive type is given by the following definition of the lists of elements of (parametric) type A , which we give in Coq notation (data types are called “Sets” in the CIC):

```
Inductive list : Set :=
| nil : list
| cons : A -> list -> list.
```

The type is defined by its constructors, in this case `nil : list A` and `cons : A -> list A -> list A` and it is understood that its elements are obtained as finite combinations of the constructors. Well-founded recursion for these types is available via the `Fixpoint` operator.

On top of this, a higher-order logic is available which serves to predicate on the various data types. The interpretation of the propositions is constructive, i.e. a proposition is defined by specifying what a proof of it is and a proposition is true if and only if a proof of it has been constructed. As a consequence, elementary predicates are also defined as inductive types, by giving the corresponding proof constructors. The type of propositions is called `Prop`.

We refer to [18,12] for further details on the CIC and Coq, respectively.

2.2 The Framework at a Glance

Although our approach is language independent, we are working with the ATL technical space. ATL (Atlas Transformation Language, [8]) is a hybrid of declarative and imperative transformation language. Since we are concerned with a model-to-model relational approach, we focused on the declarative part of ATL. In this context, an ATL transformation specification is composed of rules that define the correspondence between source and target model elements. In this technical space, source and target metamodels are specified using KM3 (Kernel MetaMetaModel, [3]) which provides a textual concrete syntax that eases the coding of metamodels.

During software construction a developer specifies the input and output metamodels and the transformation between them. We propose that at this point a separation of duties is implemented for performing formal verification. We conceive the participation of a (human, expert) verifier, who will carry out a semi-automatic translation process from the ATL to the CIC technical space, as outlined in Figure 2.

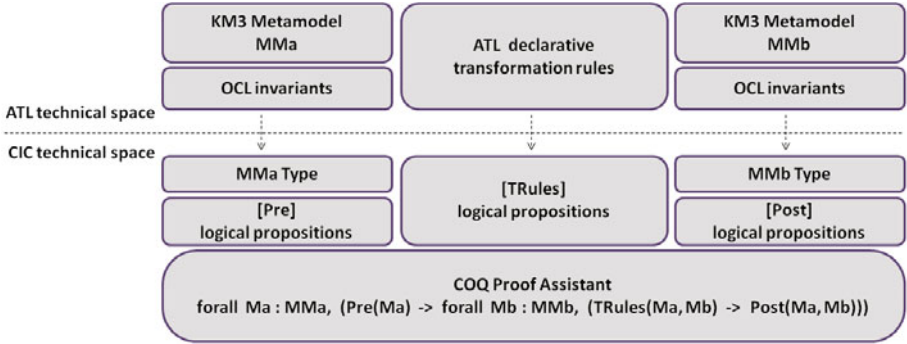


Fig. 2. An outline of our approach

The first step of the process is the formalization of the KM3 source and target metamodels as inductive types (MMA and MMb). Then, every ATL transformation rule is transformed into a logical proposition involving both the source and target metamodels (TRules). The third step is the translation of every OCL invariant into a logical proposition. Source invariants are taken as pre-conditions (Pre) of the model transformation. Target invariants and the desired properties of the transformation are taken as post-conditions (Post), which will be our proof goals. Finally, the verification is interactively performed in Coq by proving the post-conditions assuming that both the pre-conditions and the transformation rules hold. Additionally, the verifier can use the full expressiveness of Coq in order to include post-conditions that cannot be, or are not suitable to be, expressed in OCL.

2.3 A Running Example

We will illustrate our proposal by using an example based on a simplified version of the well-known Class to Relational model transformation [19]. Figure 3 shows both metamodels of this transformation. An UML class diagram consists of classes which contain one or more attributes. Each attribute has a type that is a primitive datatype. Every class, attribute and primitive data type is generalized into an abstract UML model element which contains a name and a kind (persistent or not persistent). On the other side, a RDBMS model consists of tables which contains one or more columns. Each column has a type, and every RDBMS model element is generalized into an abstract RDBMS model element.

The transformation describes how persistent classes of a simple UML class diagram are mapped to tables of a RDBMS model with the same name and kind. Attributes of the persistent class map to columns of the table. The type of the column is a string representation of the primitive data type associated to the attribute.

This example is clearly not a critical application where our approach is particularly helpful. However, it is complete and simple enough to exemplify our approach, and has been used as a standard test case for various transformation

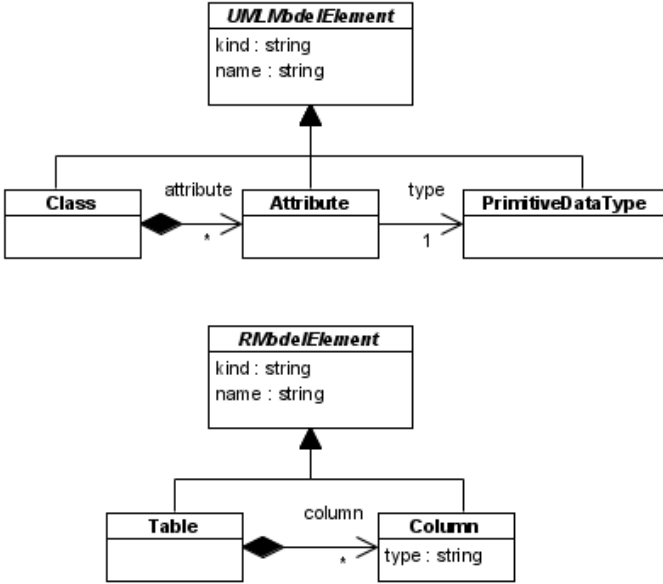


Fig. 3. UML metamodel and RDBMS metamodel

languages. For this reason, it will be used in the following sections in order to exemplify the verification process steps.

3 Formalization of Metamodels and Models

In this section we show how to represent metamodels and models in the CIC. For metamodels we show how every KM3 construction is translated into Coq notation, using the UML metamodel of the example mentioned in Figure 3 above. The whole translation has been defined and implemented as an ATL transformation. For space reasons we cannot include it here but it can be found in [20].

Data Types and Enumerations. Coq supports primitive data types like strings, booleans and natural numbers, among others, via libraries equipped with many useful functions. This is enough to represent ATL primitive types. In ATL it is also possible to define enumeration types and use them to define class attributes. Enumeration types are directly represented in Coq as inductive types with one constructor for each enumeration literal.

Classes and Attributes. A class has attributes. An attribute has a name, a multiplicity and a type. We represent classes using inductive types. For each class its attributes are represented as components of the corresponding type by means of a constructor which has its attributes as parameters. A class can be abstract, meaning that there are no direct instances of it. This impacts on the

representation of models as explained below, but not on the inductive representation of the class. In the example above, the class `UMLModelElement` is defined in Coq as follows.

```
Inductive UMLModelElement : Set :=
  | Build_UMLModelElement (oid : nat) (name : string) (kind : string)
```

Notice the presence of the component named `oid` in the representation of the `UMLModelElement`. This provides a means for identifying the actual objects that are to be instances of the various classes, i.e. the `oids` implement object identity, beyond the constructor identity provided by the CIC.

In order to manipulate the components of the classes we define projections for each attribute. They are trivially defined using pattern matching. As an example we show just the projection of the attribute `name` of the `UMLModelElement` class.

```
Definition UMLModelElement_name (o : UMLModelElement) : string :=
  match o with
  | (Build_UMLModelElement _ n _) => n
  end.
```

References. A reference represents an association between classes. It has a name, a multiplicity and a type (of the element been referenced), and it may have an opposite reference (bidirectional association). The natural choice for representing associations is by lists of pairs of related elements. In order to optimize navigability, the references of each class can be considered as components of the corresponding type, in the same way as attributes. This results in using mutually inductive types for representing related classes.

In the example below, the references from a `Class` to its `Attributes` is represented as a component within the constructor of the `Class` type.

```
Inductive Class : Set :=
  | Build_Class ...
  (attribute : list Attribute)
```

Now, if for a given class and reference an opposite reference exists, the elements of the source class must form part of those of the target class and viceversa, i.e. the objects of both classes are not well-founded. In general, this situation might arise whenever several classes are mutually related through cycling associations and we can characterize it as the admissibility of circularity in the actual construction of (thereby infinite) objects. In these cases we seem to need co-inductive types, as pointed out in [15,17]. However, taking such an approach forces us in the general case to introduce all the mutually connected classes as mutually defined co-inductive types. And then some disadvantages arise, concerning both the correctness of the representation and its ease of use. The main problem is that there will in general be cycles of references of classes of the model for which no actual cycle at the level of object formation is intended to occur, even when some other reference cycles in the same model allow the circularity of object formation.

Hence, we have in general that some of the classes involved in the represented metamodel will not be intended to actually contain infinite structures, namely those participating in references in which no actual cycle at the level of objects is admissible. But even in such cases, the definition of the classes as co-inductive allows them to contain infinite structures. This compromises the correctness of the representation in at least two respects: first, circularity at the level of objects cannot be prevented at syntax (type) level and secondly, the termination of functions on these types cannot be enforced. Although these restrictions could be imposed on the co-inductive definition of the model, that would lead to a representation too awkward to manage in practice.

We therefore decide to use only inductive types. This is enough for the cases in which no circularity at the level of objects is to be allowed, since the well-foundedness of the latter is imposed by construction. More precisely, we represent directly only unidirectional references, using mutually inductive types whenever circularity at the level of the objects is not allowed in the metamodel. If, on the contrary, we should have to allow for such circularity then we use the first representation mentioned above, i.e. associations as list of pairs of related elements. This procedure has as a particular case that of the bi-directional associations.

Deciding which references are represented in either way in an optimal way depends on how the model elements are used in the transformation. This is a point in which, although possible, the automation of the representation of the metamodel might not be desirable. In the implemented translation of [20] the references of each class are represented as components of the corresponding type, and circularity must be manually “cut” by the human verifier.

Multiplicities. Attributes and references have multiplicities. Each multiplicity has a lower and an upper value and multiplicity 1 is assumed if none is declared.

When representing references as components, multiplicity reflects itself in the type of the component. This is the same as in the case of attributes. Multiplicity $0..1$ is represented with the `option` type constructor, which has constructors `None` representing no element and `Some x` for elements `x` in the original type. If the upper multiplicity value is greater than 1, the multiplicity is represented with a (possibly ordered) list type. Multiplicity 1 corresponds just to the type of the component.

In the example we have the following multiplicities.

```
[1-1] -- name : string
[0-*] -- attribute : list Attribute
```

When representing associations as list of pairs of related elements, multiplicity is enforced by explicit constraints on the number of pairs allowed for each element of the participant classes.

Generalization and Abstract Classes. In ATL it is also possible to define generalization relations between classes. The CIC does not have a notion of subtyping between types, unlike [15]. We represent this notion as references from the subtypes to it(s) supertype(s). With this representation we can easily

navigate from a subclass to a property of its superclass. Notice that when a generalization exists, the oids are located only at the topmost supertype.

In the example, the `PrimitiveDataType` class has a reference to its supertype `UMLModelElement`.

```
PrimitiveDataType : Set :=
  | Build_PrimitiveDataType (super : UMLModelElement)
```

The whole UML metamodel in Figure 3 is defined in Coq as follows.

```
Inductive UMLModelElement : Set :=
  | Build_UMLModelElement (oid : nat) (name : string) (kind : string).

Inductive Class : Set :=
  | Build_Class (super : UMLModelElement)
    (attribute : list Attribute)
with
  Attribute : Set :=
  | Build_Attribute (super : UMLModelElement)
    (type : PrimitiveDataType)
with
  PrimitiveDataType : Set :=
  | Build_PrimitiveDataType (super : UMLModelElement).
```

Finally, a model that conforms to a metamodel is represented as a record containing the lists of instances of each non-abstract metamodel element, as suggested in [21]. In the example a model conforming to the UML metamodel would be a record of the following type.

```
Record SimpleUML : Set :=
  mkSimpleUML {classAllInstances : list Class;
    primitiveDataTypeAllInstances : list PrimitiveDataType;
    attributeAllInstances : list Attribute
  }.
```

In the example every element in the model is reachable from the `Class` instances, so the record can in fact be reduced to just a list of classes.

4 Translation of the Model Transformation

We show next how ATL constructs can be translated into Coq notation, using the example transformation in Section 2.3. Briefly, every ATL declarative transformation rule is transformed into a logical proposition, and helper (auxiliary) functions are translated into Coq functions. Not every ATL construct is considered since some of them (e.g. modules) are not relevant for our study. The whole transformation of the example can be found in [20].

Data Types. ATL's data types are based on the OCL. They include primitive types (boolean, integer, real, string), tuples, enumerates, collections (set, ordered set, bag, sequence), among others. All these types can be represented in Coq as described in section [3](#).

OCL Declarative Expressions. ATL uses additional OCL declarative expressions in order to structure the code. ATL's *If-Then-Else*, *Let* (which enables the definition of variables) and constant expressions (constant values of any supported data type) are natively supported in Coq. Finally, the collection iterative expressions are supported in Coq with recursion operators on lists.

Helpers and Attributes. Helper/attribute call expressions as well as operation call expressions are OCL-based expressions. ATL helpers factorize code that can be called from different points of an ATL transformation. An ATL helper is defined by the following elements: a name, a context type, a return value type, an ATL expression that represents the code of the ATL helper, and an optional set of parameters, in which a parameter is identified by a pair (parameter name, parameter type). From a functional point of view an attribute is a helper that accepts no parameters. Both helpers and attributes are represented as functions in the richly-typed functional programming language provided by Coq. The main issue in this translation is that Coq imposes the condition that every recursion be well-founded, which has to be proven in each case.

In the example there is the following helper function that transforms a `PrimitiveDataType` into a string which represents the type of a column in the database.

```

helper context SimpleUML!PrimitiveDataType def :
  primitiveTypeToStringType : String =
    if (self.name = 'INTEGER')
    then 'NUMBER'
    else if (self.name = 'BOOLEAN')
      then 'BOOLEAN'
      else 'VARCHAR'
    endif
  endif;

```

The Coq function resulting from its translation is as follows. Notice that the comparison between strings (=) is performed by an auxiliary function `string_eq_bool` which returns a boolean value.

```

Definition primitiveTypeToStringType (primitiveType : string) : string :=
  match (string_eq_bool primitiveType "INTEGER") with
  | true => "NUMBER"
  | false => match (string_eq_bool primitiveType "BOOLEAN") with
    | true => "BOOLEAN"
    | false => "VARCHAR"
  end
end.

```

Matched Rules. The matched rules constitute the core of an ATL declarative transformation since they make it possible to specify the kind of source elements for which target elements must be generated, and the way the generated target elements have to be initialized. A matched rule is introduced by the following construction.

```
rule rule_name {
  from in_var : in_type [(condition)]
    [using { var1 : var_type1 = init_exp1;
    ... }]
  to out_var1 : out_type1 (bindings1),
  ...
}
```

The source pattern is defined after the keyword `from`. It enables to specify a model element variable that corresponds to the type of source elements that the rule has to match. When defined, the local variable section is introduced by the keyword `using`. The target pattern of a matched rule is introduced by the keyword `to`. It serves to specify the elements to be generated when the source pattern of the rule is matched, and how these generated elements are initialized (bindings). An optional condition (expressed as an ATL expression) within the rule source pattern is used to select the subset of the source elements that conform to the matching type.

Matched rules are generally translated into propositions of the form

$$\forall a:A. (a \in InstA \wedge Cond(a) \rightarrow \exists b:B. (b \in InstB \wedge Rel(a,b)))$$

expressing that for every object a of the type A (of the source model element in the matched rule) in the set $InstA$ of all instances of type A which satisfies certain condition $Cond$, there exists an object b of the type B (of the target model element in the matched rule) in the set $InstB$ of all instances of type B , for whom the relation Rel holds. The relation Rel is a conjunction of the bindings defined in the matched rule. If there are no other matched rules that define the existence of a target model element for whom the same relation Rel holds, then the proposition must state the unique existence of the target model element. There are also propositions describing the relation in the reverse direction (i.e. from the target to the source elements).

The formulæ thus obtained amount to (basic) specifications of the transformation rules at a propositional level. This stands in contrast to the approach in [17] where transformations were represented as functions about which the relevant properties had to be proven, leading to lengthy work that can now be avoided.

In the example we have the following matched rule that transforms an **Attribute** of a class diagram into a **Column** of the database. The name of the column will be the name of the attribute, and the type of the column will be the name of the **PrimitiveDataType** associated to the attribute (the helper already introduced is used in this case).

```

rule AttributeToColumn{
  from a : SimpleUML!Attribute ()
  to c : SimpleRDBMS!Column (
    name <- a.name,
    type <- a.type.primitiveTypeToStringType
  )
}

```

We represent this matched rule as follows:

```

Definition AttributeToColumn (c : Class) (t : Table) : Prop :=
  (forall atr:Attribute, In atr (Class_attribute c) ->
    exists! col:Column, In col (Table_column t) /\
      RModelElement_name (Column_super col) = Attribute_name atr /\
      Column_type col = primitiveTypeToStringType
        (PrimitiveDataType_name (Attribute_type atr)))
  /\
  (forall col:Column, In col (Table_column t) ->
    exists! atr:Attribute, In atr (Class_attribute c) /\
      RModelElement_name (Column_super col) = Attribute_name atr /\
      Column_type col = primitiveTypeToStringType
        (PrimitiveDataType_name (Attribute_type atr))).

```

Notice that the relation is described in both directions, and also that there is only one source and target elements for which the proposition holds.

For the sake of completeness we present the other matched rule that transforms every persistent `Class` into a `Table` of the database. The name of the table must be the same as the class, and the columns of the table will be the transformation of the attributes of the class which is performed by the matched rule `AttributeToColumn`.

```

rule ClassToTable{
  from c : SimpleUML!Class (c.kind = 'Persistent')
  to t : SimpleRDBMS!Table (
    name <- c.name,
    cols <- c.attribute
  )
}

```

This matched rule is represented in Coq as follows.

```

Definition ClassToTable (ma : SimpleUML) (mb : SimpleRDBMS) : Prop :=
  (forall c:Class, In c (MClass_classAllInstances ma) /\
    Class_kind c = "Persistent" ->
    exists! t:Table, In t (MRelational_tableAllInstances mb) /\
      Class_name c = Table_name t /\
      AttributeToColumn c t)
  /\

```

```
(forall t:Table, In t (MRelational_tableAllInstances mb) ->
  exists! c:Class, In c (MClass_classAllInstances ma) /\
    Class_kind c = "Persistent" /\
    Class_name c = Table_name t /\
    AttributeToColumn c t).
```

5 Verification of Properties

OCl invariants of both source and target metamodels are translated into propositions in the CIC. This, at a large measure, can be done automatically following the ideas presented in [22]. The desired properties of the transformation are specified in the CIC by the verifier using the full potential of the logic. These properties will in general establish relations between (any instances of) the source and target metamodel connected by the transformation.

A simple property of the example transformation is that the length of the `Columns` within a `Table` must be greater than zero. This can be written in OCL as follows.

```
context Table inv:
  self.column->length() > 0
```

In Coq, this property can be expressed as follows.

```
Definition TableAtLeastOneCol (model : SimpleRDBMS) : Prop :=
  forall t:Table, (In t (MRelational_tableAllInstances model)) ->
    length (Table_column t) > 0.
```

This property holds by the fact that every `Attribute` is transformed into a `Column` and that every `Class` has at least one `Attribute`. This information is given in the transformation rules and in the source invariants, respectively.

The invariants of the target metamodel and any other desired properties of the transformation (*Post*) are interactively verified in Coq by assuming that the invariants of the source metamodel (*Pre*), and the transformation rules (*TRules*) hold. In this way, the correctness proposition becomes:

$$\forall Ma:MMa. (Pre(Ma) \rightarrow \forall Mb:MMb. (TRules(Ma,Mb) \rightarrow Post(Ma,Mb)))$$

In the example, the Coq lemma to prove is as follows.

```
Definition Post (ma : SimpleUML) (mb : SimpleRDBMS) : Prop :=
  TableAtLeastOneCol mb.
```

```
Definition TRules (ma : SimpleUML) (mb : SimpleRDBMS) : Prop :=
  ClassToTable ma mb.
```

```
Lemma Cert_Class2Relational:
  forall ma:SimpleUML, Pre ma -> forall mb:SimpleRDBMS, TRules ma mb
  -> Post ma mb.
```

Notice that in this case the transformations rules are only the satisfaction of the `ClassToTable` rule, since every source model element involved in the transformation is reached from the class elements. The postcondition is the property `TableAtLeastOneCol`, defined above.

The Coq proof assistant helps building proofs using tactics (inference rules). We refer to the Coq documentation [13] for further details. The proof of this property can be found in [20].

There are other properties which can be proved for this transformation, for example the following OCL invariants.

```
context Table inv:
  Table.allInstances()->isUnique(name)

context Table inv:
  self.cols->isUnique(name)
```

The first one states that the name of a `Table` is unique. This holds because every `Class` is transformed into a `Table` and because the name of a `Class` is unique. The second property states that the name of a `Column` is unique within a `Table`, which holds because the name of an `Attribute` is unique within a `Class`.

The framework allows stating and proving more interesting properties which involve both the source and target metamodels. In these cases the properties cannot be expressed in OCL but can be expressed in Coq. For example, we proved that the number of tables is equal to the number of persistent classes, in Coq notation:

```
Definition ClassTableEqLen (ma : SimpleUML)
  (mb : SimpleRDBMS) : Prop :=
  length (filter isPersistent (MClass_classAllInstances ma)) =
  length (MRelational_tableAllInstances mb).
```

The proof of this property is done by induction on both `MClass_classAllInstances` and `MRelational_tableAllInstances` which are the lists of all the instances of type `Class` and `Table`, respectively. This proof can also be found in [20]. In a similar way, we can prove that the number of `Columns` within any `Table` is equal to the number of `Attributes` of the corresponding `Class`.

6 Conclusions and Further Work

We have described a type-theoretic framework that allows the full formal verification of model transformations, at a metamodel level, and considering any kind of transformations and properties. We have proposed a separation of duties between developers and verifiers, based on a semi-automatic translation process switching from the ATL to the CIC (Calculus of Inductive Constructions) technical space as implemented on Coq. Within this framework, source and target metamodels (*MMa* and *MMb*) are represented as inductive types, and each

transformation rule of the model transformation Mt is represented as a logical formula of $\forall\exists$ form stating that for every model element satisfying a certain (pre-)condition, there exists a target model element which stands in the relation specified by the transformation rule with the source model element ($TRule$). The correctness of the model transformation is stated by a formula.

$$\forall Ma:MMa. (Pre(Ma) \rightarrow \exists Mb:MMb. (TRules(Ma,Mb) \rightarrow Post(Ma,Mb)))$$

where Pre is a translation of the source invariants, $TRules$ is the conjunction of the transformation rules, and $Post$ is a translation of the target invariants plus any other desired property to be proved. A proof of this formula ensures that the transformation rules satisfy the target invariants as well as the desired properties.

The translation into Coq of the KM3 metamodels can be performed fully automatically. On the other hand, at the moment the verifiers have to deal with: references in the metamodels that must be “cut” to avoid circularity in an optimal way, the translation of the OCL invariants –which can at a large measure be done automatically– and the translation of the ATL transformation rules and helpers. Then he can proceed to perform the formal verification.

As a proof of concepts, we have applied our approach to a simplified version of the well-known Class to Relational model transformation broadly studied in the literature [19]. The resulting Coq code can be found in [20].

With this approach we lose full automation to gain in return strength of the achieved results. We think the approach could be particularly helpful in proving the existence of zero-fault model transformations within the development of critical systems.

So far the non-automatic parts of the process of translation involved in our proposal can in general be carried out directly enough to indeed provide increased confidence in the outcome.

We are currently working on setting up a semantics of transformation languages in type theory which will lead to a greater automatic capability at the level of the framework, particularly concerning the outlined method for translating transformations.

Our medium-term goals are the full development of the framework and its integration with ATL and Coq. In the long-term we will work on simplifying the proof process. In this direction we aim at generating auxiliary libraries with proofs of basic properties and also work on proof patterns detection in order to improve the facility of use of the proof assistant.

Acknowledgement

This work has been partially funded by the National Research and Innovation Agency (ANII) of Uruguay through the “Verification of UML Based Behavioral Model Transformations” project [20].

References

1. Kent, S.: Model-Driven Engineering. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. LNCS, vol. 2335, pp. 286–298. Springer, Heidelberg (2002)
2. OMG: Meta Object Facility (MOF) 2.0 Core Specification. Object Management Group, Specification Version 2.0 (2003)
3. ATLAS Group: Kernel MetaMetaModel. LINA & INRIA. Manual v0.3 (2005)
4. OMG: UML 2.0 Object Constraint Language. Object Management Group, Specification Version 2.0 (2006)
5. Mens, T., Czarnecki, K., van Gorp, P.: A Taxonomy of Model Transformation. ENTCS, vol. 152, pp. 125–142. Springer, Heidelberg (2006)
6. Czarnecki, K., Helsen, S.: Feature-Based Survey of Model Transformation Approaches. IBM Systems Journal 45(3), 621–645 (2006)
7. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation. Object Management Group, Specification Version 1.0 (2008)
8. ATLAS Group: Atlas Transformation Language. LINA & INRIA. User Guide (2009)
9. Anastasakis, K., Bordbar, B., Küster, J.M.: Analysis of Model Transformations via Alloy. In: Proc. 4th Workshop on Model-Driven Engineering, Verification and Validation, pp. 47–56 (2007)
10. Pons, C., García, D.: A Lightweight Approach for the Semantic Validation of Model Refinements. ENTCS, vol. 220, pp. 43–61. Springer, Heidelberg (2008)
11. Giese, H., et al.: Towards Verified Model Transformations. In: Proc. 3rd International Workshop on Model Development, Validation and Verification, pp. 78–93 (2006)
12. Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer, Heidelberg (2004)
13. The Coq Development Team: The Coq Proof Assistant: Reference Manual (2009)
14. Leroy, X.: Formal Verification of a Realistic Compiler. Commun. ACM 52, 107–115 (2009)
15. Poernomo, I.: A Type Theoretic Framework for Formal Metamodelling. In: Reussner, R., Stafford, J.A., Ren, X.-M. (eds.) Architecting Systems with Trustworthy Components. LNCS, vol. 3938, pp. 262–298. Springer, Heidelberg (2006)
16. Poernomo, I.: Proofs-as-Model Transformations. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 214–228. Springer, Heidelberg (2008)
17. Calegari, D., Luna, C., Szasz, N., Tasistro, A.: Experiment with a Type-Theoretic Approach to the Verification of Model Transformations. In: Proc. 2nd Chilean Workshop on Formal Methods, pp. 29–36 (2009), http://jcc2009.usach.cl/?page_id=631 (last visit: August 2010)
18. Coquand, T., Paulin, C.: Inductively Defined Types. In: Martin-Löf, P., Mints, G. (eds.) COLOG 1988. LNCS, vol. 417, pp. 50–66. Springer, Heidelberg (1990)
19. Bézivin, J., Rumpe, B., Schürr, A., Tratt, L.: Model Transformations in Practice Workshop. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 120–127. Springer, Heidelberg (2006)
20. Verification of UML-Based Behavioral Model Transformations Project, <http://www.fing.edu.uy/inco/grupos/coal/field.php/Proyectos/ANII09> (last visit: August 2010)
21. Steel, J., Jézéquel, J.M.: On Model Typing. SoSyM 6, 401–413 (2007)
22. Beckert, B., Keller, U., Schmitt, P.: Translating the Object Constraint Language into First-Order Predicate Logic. In: Workshop at Federated Logic Conferences (2002)

Simulating Truly Concurrent CSP

Moritz Kleine¹ and J.W. Sanders²

¹ Software Engineering Group

Technische Universität Berlin, Germany

² International Institute for Software Technology

United Nations University, Macao SAR China

Abstract. Process algebras like CSP provide a convenient intermediate-level formalism for the design of concurrent systems by allowing processes to be combined in parallel in such a way that the designer abstracts synchronization mechanisms and simultaneity of events. However some purposes require potential simultaneity to be made explicit. One approach is to produce new semantics models encapsulating that information. The approach taken here is to use the standard models and the CSP tool, FDR, to simulate a process in such a way to reveal potentially-simultaneous events. The simulation is achieved by a construction that splits events into start and end events and monitors the result in a manner faithful to the original process. The method is applied to determine pairs of possibly concurrent events and to compute maximal simultaneity in a CSP design.

1 Introduction

Process algebras, like CSP [4,9], allow synchronizing processes to be combined in parallel with the result that the system designer need not be concerned about exploiting simultaneity, which may arise naturally in an implementation conforming to the design. But sometimes, for example for purposes of simulation, it is useful to know what potential simultaneity a design embodies, and then the abstraction carefully built in to the process algebra must be revoked.

The purpose of this paper is to study one way to make explicit the simultaneity of events implicit in a CSP process. Events were designed in CSP to be instantaneous, on the understanding that duration can then be modelled by splitting an event into start and end events. Our approach starts by unravelling that assumption, and proceeds by constructing a faithful, controlled simulation of the process. The construction is shown to be faithful in the sense that the simulated version equals the original process in the traces semantics. The approach is shown at work on three small but typical examples. It is defined for a subset of CSP_M [10] for subsequent analysis with FDR [9] or ProB [8]. Although our approach is not limited to finite-state processes, applications here are to finite-state processes analyzed using FDR or ProB.

Several applications of the approach are discussed. For purposes of motivation, observe that CSP events, being instantaneous, abstract the actions or whole procedures of a lower-level programming language. Of course the advantage is simplicity

of reasoning about things like data values and execution of procedures. Examples of systems that allow execution of user-defined functions when an event occurs are CSP-OZ [1], CSP++ [2] and the Process Analysis Toolkit PAT [3]. Such systems separate execution of user-defined functions from one another (they are not executed in parallel) or disallow user-defined functions on synchronization events. Those assumptions limit the gains promised by concurrency.

An important consequence of the approach taken here is that sets of events that may occur concurrently are computed statically. As a result, the approach appears suitable for realistic simulation (in the sense that functions attached to events may be executed concurrently but may require positive duration), overcoming the limitations mentioned in the previous paragraph. Indeed any event (even synchronization events) can be linked to a terminating user-defined function, and the functions executed concurrently. A user-defined function is started immediately after its corresponding start (non-tick) event and after its termination the corresponding end event becomes available.

Sect. 2 provides an overview of CSP and introduces its tools required in this paper. Sect. 3 introduces the transformation T that performs splitting of events, then Sect. 4 shows how the transformed process can be used to compose a system that simulates the original process while collecting concurrency information. Sect. 5 presents properties used to prove that the construction preserves the semantics of the original process. Three examples are presented in Sect. 6, though they may be browsed earlier. Limitations, related and further work are discussed in Sect. 7. The conclusion is presented in Sect. 8.

2 CSP

The process calculus Communicating Sequential Processes (CSP), introduced by Hoare in the late 1970s, was largely stable by the mid 1980s [4] since when it has been widely applied and developed. Its strength is the specification and verification of reactive and concurrent systems in which synchronization and communication play a key role. Its *processes* perform *events* that are both atomic and instantaneous. The set of events that may be communicated by a process is said to comprise its alphabet Σ . If a process offers an event with which its environment agrees to synchronize, the event is performed. A sequence of events that a process may perform is called a *trace* of the process. For (semantic) convenience the alphabet of each process is extended to contain two further events: $\tau \notin \Sigma$ represents an internal event and $\surd \notin \Sigma$ represents termination. Processes are patterns of computation observable, in the standard model, by the sets (because of nondeterminism) of events on which they may refuse to synchronize at any point in their evolution, as determined by the trace that has been performed; the trace/set pair is called a *failure*. A process can be thought of as a (possibly infinite) transition system.

CSP is equipped with a rich set of process operators including prefixing ($a \rightarrow P$), external choice ($P \square Q$), internal choice ($P \sqcap Q$), sequential composition ($P ; Q$), abstraction or hiding ($P \setminus A$) and parallel composition ($P \mid_A Q$).

The process $a \rightarrow P$ offers its environment the opportunity to synchronize on a in which case it then behaves like P . The process $P \square Q$ offers its environment a choice between P and Q based on synchronization with their initial events. The process $P \sqcap Q$ behaves like either P or Q but the choice is made internally, beyond environmental influence. A third kind of choice, timeout $P \triangleright Q$, (sometimes called ‘sliding choice’) combines external and internal influences. It is represented using internal and external choice by $P \triangleright Q = (P \square Q) \sqcap Q$. The process $P ; Q$ behaves like P and, if that terminates, then behaves like Q . The process $P \setminus A$ executes the events in the set A internally, without synchronization by its environment; they can be thought of as being replaced by τ events. The parallel composition $P \mid_A Q$ requires P and Q to synchronize on each event $a \in A$, but performs other events of P or Q as determined by those processes. *STOP* and *SKIP* are ‘atomic’ processes; the former models deadlock by offering no events; the latter models successful termination by offering only \checkmark . For example the following process, defined as an abstraction of components that synchronize on a , simply offers the event a and then terminates successfully.

$$(a \rightarrow \text{SKIP} \mid_{\{a\}} (b \rightarrow \text{SKIP} ; a \rightarrow \text{SKIP})) \setminus \{b\}$$

CSP has a range of semantic models, the most basic of which are: the traces model \mathcal{T} ; the stable failures model \mathcal{F} ; and the failures-divergences model \mathcal{FD} [9]. In the traces semantics of CSP, a process is represented by the set of all its possible traces; the result is useful in specifying safety properties. The stable failures semantics, by recording the refusals of a process, is useful in specifying a system’s safety and liveness properties. In addition to traces and refusals, infinite sequences of internal transitions (τ events), called *divergences*, are required. Divergence can be introduced by hiding, the symbol *div* and by ill-formed recursion like $P = P$. Thus, for example, internal and external choice are indistinguishable in the traces model but resolved in the stable failures model, whilst divergence is resolved in the failures-divergences model.

Conformance in CSP is expressed by *refinement*. Informally, $P \sqsubseteq Q$ means that Q conforms to P , or that Q ’s behaviors are contained in those of P . Formally, for any of the three semantic models $\mathcal{M} \in \{\mathcal{T}, \mathcal{F}, \mathcal{FD}\}$,

$$P \sqsubseteq_{\mathcal{M}} Q \Leftrightarrow \mathcal{M}[Q] \subseteq \mathcal{M}[P]$$

where $\mathcal{M}[P]$ denotes the semantics of process P in semantic model \mathcal{M} . The result is that CSP provides a refinement calculus supporting process development from abstract specification to implementation.

Algebraic reasoning is supported by refinement laws that correspond to containments in the failures-divergences model. Examples are: idempotency of internal choice, $P \sqcap P = P$; distributivity of internal choice over parallel composition, $(P \mid_A Q) \sqcap R = (P \sqcap R) \mid_A (Q \sqcap R)$; and resolution of nondeterminism, $P \sqcap Q \sqsubseteq P$. Using these laws, the timeout operator can also be written $P \triangleright Q = (P \sqcap \text{STOP}) \square Q$.

All three refinements are supported by the automatic refinement checker FDR [3](#) which proves or refutes assertions of the form $P \sqsubseteq_{\mathcal{M}} Q$. FDR inputs processes expressed in CSP_M , which is now the *de facto* standard for machine-readable CSP. CSP_M expresses CSP by a small but powerful functional language, offering constructs such as *lambda* and *let* expressions and supporting pattern matching and currying. It also provides a number of predefined data types, including Booleans, integers, sequences and sets, and allows user-defined data types. The global event set is defined by the set of typed channel declarations of a CSP_M script. The $\{|\cdot|\}$ -operator can be used to compute the set of events that complete the set of given prefixes (e.g. channels). CSP_M models can also be animated and verified by LTL model checking using ProB [8](#).

3 The Transformation T

In this section we present the transformation T that achieves the simulation motivated in the Introduction. T models duration of events by splitting them. Hidden transitions are exposed by introducing fresh events. The purpose of the transformation is to put the transformed process $T(P)$ into parallel with a control component that records the start and end events of every transition of the original process P (including hidden events). The control component can then record possible simultaneity in T and thus be used to compute possible concurrency in P .

Throughout this paper processes are expressed using the syntax

$$(x : X \rightarrow P(x)) \mid P ; Q \mid P \square Q \mid P \mid_A Q \mid P \setminus A \mid P \sqcap Q \mid P[M] \mid P \triangleright Q$$

for general choice, sequential composition, external choice, parallel composition, hiding, internal choice, renaming and timeout respectively. Recall that general choice includes the atomic processes and prefixing, by appropriate choice of X . Renaming of event x to y in process P is written $P[x \leftarrow y]$, or more generally $P[M]$ where M is a mapping from source events to target events. Recall that interleaving (\parallel) is the special case of parallel composition without synchronization ($\{\emptyset\}$) in the interleaving semantics of CSP.

Let s, sh, e, eh be fresh channels relative to Σ_P . The transformation T is defined as follows.

$$\begin{aligned} T(x : X \rightarrow P(x)) &= s.x : \{s.y \mid y \in X\} \rightarrow e.x \rightarrow T(P(x)) \\ T(P \otimes Q) &= T(P) \otimes T(Q), \quad \text{for } \otimes = ; \text{ and } \otimes = \square \\ T(P \mid_A Q) &= T(P) \mid_{\{s.x, e.x \mid x \in A\}} T(Q) \\ T(P \setminus A) &= T(P)[s.x \leftarrow sh.x, e.x \leftarrow eh.x \mid x \in A] \\ T(P \sqcap Q) &= sh.ic_i \rightarrow eh.ic_i \rightarrow (T(P) \sqcap T(Q)) \\ T(P[M]) &= T(P)[s.x \leftarrow s.y, e.x \leftarrow e.y \mid (x, y) \in M] \\ T(P \triangleright Q) &= T(P) \square (sh.to_i \rightarrow eh.to_i \rightarrow T(Q)) \end{aligned}$$

For general choice, T splits each event x into its start event $s.x$ and end event $e.x$. As special cases,

$$\begin{aligned} T(P) &= STOP \\ T(SKIP) &= SKIP \\ T(x \rightarrow P) &= s.x \rightarrow e.x \rightarrow T(P). \end{aligned}$$

T distributes over sequential composition, external choice and parallel composition, in the latter case by synchronizing on the split events instead of the original events. For hiding, T communicates the split events over the channels sh and eh (standing for ‘start hidden’ and ‘end hidden’, respectively). For each internal choice, thought of as resulting from an internal transition, T introduces a fresh hidden event labelled ic_i for that transition. It is then split and communicated over the sh and eh channels responsible for hidden events. T distributes over renaming by lifting the renaming to the split events. Finally T distributes over the operands of a timeout, replaces the timeout by external choice and introduces a fresh timeout event labelled to_i for each timeout communicated over the channels sh and eh responsible for hidden events (recall the motivation for the timeout operator to express the abstraction of one initial event in an external choice; see, for example, [9]).

The alphabets of P and $T(P)$ are disjoint, since

$$\Sigma_{T(P)} \subseteq \{s, e, sh, eh\}.$$

Equality (in the traces model) with the original process P is established by hiding the end events of the original visible transitions and all events corresponding to hidden transitions and renaming the start events of the original transition back to their source names. The exposition of hidden transitions disallows equality in the failures (and in the failures-divergences) model, because in general

$$(P \square Q) \setminus A \neq P \setminus A \square Q \setminus A$$

in these models, but

$$(P \square Q) \setminus A =_T P \setminus A \square Q \setminus A.$$

Thus, there is no construction F using parallel composition, hiding and renaming such that $F(T(P \square Q)) = P \square Q$. In the next section we present a construction that preserves the traces of the original process and reveals its internal concurrency.

4 Assembling the System

The ‘control process’ to be placed in parallel with a transformed process is defined in terms of a parameter X denoting a bag; bag union is denoted \uplus , bag difference \ominus , and bag comprehension is written $\llbracket x_0, \dots, x_n \rrbracket$. Initially the bag is

empty. Let $term$ be a fresh event modeling the possibility to synchronize before successful termination.

$$\begin{aligned}
 C(X) = & \quad s?x \rightarrow C(X \uplus \llbracket x \rrbracket) \\
 & \quad \square sh?x \rightarrow C(X \uplus \llbracket x \rrbracket) \\
 & \quad \square e?x \rightarrow C(X \uplus \llbracket x \rrbracket) \\
 & \quad \square eh?x \rightarrow C(X \uplus \llbracket x \rrbracket) \\
 & \quad \square term \rightarrow SKIP
 \end{aligned} \tag{1}$$

As outlined in the previous section the process $C(X)$ is to be put in parallel with the transformed process. We define $S_{Con} = \{ | s, sh, e, eh, term | \}$ to be the alphabet on which the transformed process $T(P)$ and the controller synchronize. Now because $term \in S_{Con}$ but $term \notin \Sigma_{T(P)}$, the controller C cannot terminate while $T(P)$ is still active. Since we aim at establishing traces-equality, the parallel composition must be able to terminate if P terminates successfully. Thus, because of the Ω semantics (see [9]) and distributed termination of parallel composition, $T(P)$ is sequentially composed with the process $term \rightarrow SKIP$ in the following construction. The construction starts by transforming an input process P to $T(P)$ and combining the result with the control process (II) to achieve the result $Con(P)$ (standing for *controlled*).

$$Con(P) = (T(P); term \rightarrow SKIP) |_{S_{Con}} C(\llbracket \rrbracket) \tag{2}$$

The events $s.x$ are renamed to x , and the events in $H_{Hr} = \{ | sh, e, eh, term | \}$ are hidden using Hr (standing for *hidden and renamed*)

$$Hr(P) = P \setminus H_{Hr} [s.x \leftarrow x \mid x \in \Sigma_P] \tag{3}$$

resulting in a process

$$Ext(P) = Hr(Con(P)) \tag{4}$$

(the *extension* of P) that we think of as simulating P but enabling it to benefit from true concurrency. While concurrency cannot be distinguished from choice in the interleaved semantic models \mathcal{T} , \mathcal{F} and \mathcal{FD} , the transformation T allows them to be distinguished because start and end events of two events x and y may interleave only if x and y are concurrent in P . Thus, using the construction given above, possible concurrency is captured by the bag X maintained by the controller process $C(X)$. Whenever the size of the bag exceeds one, it holds the names of events that are performed concurrently at that point.

To ensure that a process is not ‘corrupted’ by Ext we must show that it is a fixed point of Ext in the traces semantics:

Theorem 1. *For each process P of the form given above,*

$$P =_{\mathcal{T}} Ext(P).$$

The intuition is that *Ext* splits each event in its argument, relabels the start event back to the original and hides the end event and does so whilst faithfully translating the process combinators.

A proof by structural induction is given in our technical report [6], using the results of the next section, which enforce the intuition and enable the proof to proceed uniformly.

The information stored in the bag held by C can be exploited in various ways. One obvious way is to introduce guards g_0, \dots, g_3 as follows.

$$\begin{aligned}
C_0(X) = & \quad g_0(X) \ \& \ s?x \rightarrow C_0(X \uplus \llbracket x \rrbracket) \\
& \quad \square \ g_1(X) \ \& \ sh?x \rightarrow C_0(X \uplus \llbracket x \rrbracket) \\
& \quad \square \ g_2(X) \ \& \ e?x \rightarrow C_0(X \uplus \llbracket x \rrbracket) \\
& \quad \square \ g_3(X) \ \& \ eh?x \rightarrow C_0(X \uplus \llbracket x \rrbracket) \\
& \quad \square \ term \rightarrow SKIP
\end{aligned}$$

Since the guards restrict the behavior of C_0 relative to C , we have $C \sqsubseteq_{\mathcal{T}} C_0$. Modifying the extension *Ext* of P to use C_0 yields

$$Ext_0(P) = Hr((T(P); term \rightarrow SKIP) |_{S_{con}} | C_0(\llbracket \cdot \rrbracket)).$$

By construction of C_0 and Theorem 1 we have

$$P \sqsubseteq_{\mathcal{T}} Ext_0(P).$$

In the case that checking with FDR fails to establish the refinement

$$Ext_0(P) \sqsubseteq_{\mathcal{T}} P,$$

it provides a counterexample leading to a state violating g_i .

5 Properties

In this section, t, u are sequences, the concatenation of sequences t and u is written $t \hat{\ } u$, $t \upharpoonright X$ restricts t to the elements that are contained in X and $\#t$ denotes the length of the sequence t . Containment of an element x in a sequence t is written $x \in t$. The results of this section are used in proving Theorem 1. Proofs are largely routine.

First observe the following relations between s and e events.

Proposition 1 (*s-e-precedence*). *Each instance of an event $e.x : \Sigma_{T(P)}$ is preceded by its corresponding instance $s.x : \Sigma_{T(P)}$:*

$$\forall t \hat{\ } \langle e.x \rangle : traces(T(P)) \cdot \#(t \upharpoonright \{s.x\}) > \#(t \upharpoonright \{e.x\}).$$

Proposition 2 (e-non-refusability). *After event $s.x$ has occurred in $T(P)$, no event $e.x$ can be refused until $e.x$ has occurred:*

$$\forall (t \wedge \langle s.x \rangle \wedge u, R) : \mathcal{F}(T(P)) \cdot e.x \not\text{in } u \Rightarrow e.x \notin R.$$

Traces are normally used to follow the evolution of a process using the ‘after’ operator. Here we instead use failures, so that instances of events are identified uniquely; as a result the resolution of nondeterministic choices can be observed directly, without introducing a τ operator. For example, evolution of the process $(a \rightarrow STOP \sqcap b \rightarrow STOP)$ to $a \rightarrow STOP$ is recorded as evolution from

$$\{(\langle \rangle, \{\}), (\langle \rangle, \{a\}), (\langle \rangle, \{b\})\} \quad \text{to} \quad \{(\langle \rangle, \{\}), (\langle \rangle, \{b\})\}.$$

Now instances of two events x and y are said to be possibly *conflicting* in P if they are exclusive at some point (determining the instance) of P ’s evolution. For example, the events a and b are conflicting in the initial state of $a \rightarrow STOP \sqcap b \rightarrow STOP$. The following weak definition of conflict freedom captures this intuition.

Definition 1 (Conflict freedom). *Events x and y of P are conflict-free at the point determined by $(t, R) : \mathcal{F}(P)$ in P if*

$$\forall (t \wedge \langle x \rangle, R') : \mathcal{F}(P) \cdot y \notin R' \Rightarrow y \notin R'.$$

Thus the translation T ensures:

Proposition 3 (s-e-conflict freedom). *The s and e events are conflict-free throughout $T(P)$:*

$$\forall (t, R), (t \wedge \langle e.x \rangle, R') : \mathcal{F}(T(P)) \cdot (R \cap \{|s|\}) \subseteq R'.$$

If two non-conflicting events are available at some point then they may be performed concurrently. This is captured by the following proposition.

Definition 2 (Possible concurrency). *Let $\Sigma_x = \Sigma \setminus \{e.x, eh.x\}$, $\bar{s} \in \{s, sh\}$, $e(s) = e$ and $e(sh) = eh$. The set of all possibly concurrent events of P is defined:*

$$\begin{aligned} \text{conc}(P) = \{ & (x, y) \mid \exists t \wedge \langle \bar{s}.x \rangle \wedge u \wedge \langle e(\bar{s}).x \rangle : \text{traces}(T(P)), u : (\Sigma_x)^* \cdot \\ & \bar{s}.y \text{ in } u \vee e(\bar{s}).y \text{ in } u \}. \end{aligned}$$

Then event $y : \Sigma_P$ is said to be possibly concurrent with event $x : \Sigma_P$ in P iff $(x, y) \in \text{conc}(P)$.

To expose the concurrency information recorded by the controller C we enhance the construction presented so far. We introduce a new channel $co : \text{bag } \Sigma_P$ to communicate the recorded concurrency information and modify the processes C , Con and Ext as follows.

$$\begin{aligned}
C_1(X) &= co.X \rightarrow (s?x \rightarrow C_1(X \uplus \llbracket x \rrbracket)) \\
&\quad \square sh?x \rightarrow C_1(X \uplus \llbracket x \rrbracket) \\
&\quad \square e?x \rightarrow C_1(X \uplus \llbracket x \rrbracket) \\
&\quad \square eh?x \rightarrow C_1(X \uplus \llbracket x \rrbracket) \\
&\quad \square term \rightarrow SKIP) \\
Con_1(P) &= (T(P); term \rightarrow SKIP) |_{S_{con}} | C_1(\llbracket \cdot \rrbracket) \\
Ext_1(P) &= Hr(Con_1(P))
\end{aligned}$$

We observe that: (a) hiding of co does not cause divergence because there are no adjacent co events in any trace of $Ext_1(P)$; and (b) co is not in conflict with any event in C_1 , so hiding of co does not introduce nondeterminism and consequently changes neither the failures of $Ext_1(P)$ nor its traces.

Hence

$$C_1(X) \setminus \{|co|\} = C(X)$$

and

$$Ext_1(P) \setminus \{|co|\} = Ext(P).$$

The controller process C never refuses any events offered by $T(P)$ thus, again, the following proposition holds by construction of Con .

Proposition 4. *The controller process C records the concurrent events of P :*

$$\forall (x, y) : conc(P), \exists t \cap \langle co.b \rangle : traces(Con_1(P)) \cdot x \text{ in } b \wedge y \text{ in } b.$$

The controller C is designed to record concurrency information but not change the behavior of the transformed process $T(P)$. This is captured formally by the following lemma:

Lemma 1. *For any process P as above,*

$$Ext(P) = Hr(T(P)).$$

Proof. The proof follows by ‘reduction of parallel composition’. By definition of parallel composition, $P |_{\Sigma_Q} Q = P$ if Q never refuses an event $x : \Sigma_Q$.

$$\begin{aligned}
&Ext(P) \\
&= \hspace{20em} \text{definition of } Ext \\
&Hr(Con(P))
\end{aligned}$$

$$\begin{aligned}
 &= && \text{definition of } Con \\
 &Hr((T(P); term \rightarrow SKIP) |_{S_{Con}} C(\llbracket \rrbracket)) \\
 &= && \text{reduction of parallel composition: } C(\llbracket \rrbracket) \text{ never refuses } x : S_{Con} \\
 &Hr(T(P); term \rightarrow SKIP) . \\
 &= && term \in H_{Hr} \\
 &Hr(T(P)) .
 \end{aligned}$$

□

The exposition of hidden transitions is important from a practical point of view because it allows the assignment of user-defined functions to computations that are (although present) not observable from outside the computing system, enabling the determination of events concurrent with those hidden transitions. On the theoretical side, this decision restricts equality of the source process P with its extended version $Ext(P)$ to the traces model because hiding does not distribute over external choice if initial events are affected. Neither does hiding distribute over parallel composition if the hidden alphabet intersects with the synchronization set. The following lemma shows that parallel e events can safely be removed from the synchronization sets in T so that the two sets no longer intersect, and distributivity of hiding over parallel composition holds in T .

Lemma 2. *Writing $A' = \{s.x, e.x \mid x \in A\}$ and $A'' = \{s.x \mid x \in A\}$,*

$$Hr(T(P) |_{A'} T(Q)) = Hr(T(P) |_{A''} T(Q)) .$$

Proof. The proof follows by s-e-precedence (1), e-non-refusability (2) and s-e-conflict-freedom (3).

Firstly, the equality holds trivially if $A = \emptyset$; thus assume $A \neq \emptyset$. By (1), any $e.x$ event is preceded by (a not necessarily adjacent) $s.x$ event; thus we focus on s events. Since the result does not affect $x \notin A$, we can further restrict attention to $s.x, x : A$. Thus assume $s.x, x : A$ have occurred on both sides. By definition of T , neither side can refuse $s.y, (x, y) \in conc(P |_A Q)$. So assume both sides have performed $t : \{s.x \mid x \in A\}^*$. By proposition 2, neither side can refuse any of $e.x, s.x \in t$, whilst by proposition 1, any $e.x, s.x$ in t is refused. Furthermore, by proposition 3, no $e.x$ event conflicts with any of the subsequent s events. Thus, synchronizing on $e.x, x : A$ does not affect subsequent s events. □

Finiteness of P is not required in the proof. One might suspect that hiding of e events might introduce divergence, but this could happen only if there were unbounded sequences of e events. Such sequences could be introduced by a cycle of e events but the construction of T prevents cycles of e events. Furthermore, processes such as $\mu P \cdot (x \rightarrow SKIP |_{\emptyset} P)$, although infinite state, cannot perform unbounded sequences of e events either. The s-e-precedence property asserts that each infinite sequence of e events must be preceded by an infinite sequence of s events. But unbounded sequences of s events would prevent the occurrence of any e event; so whenever e events are observable there cannot have been unbounded sequences of s events in $T(P)$; so divergence cannot occur.

6 Examples

In this section the approach is applied to three examples: (a) the example used to motivate [7]; (b) a one-place buffer from [3]; and (c) a modification of the dining philosophers from [3]. Due to technical limitations of ProB and FDR, the typing and naming scheme used in the example scripts (see [6]) differ slightly from the models presented here. For example, the controller maintains a list instead of a bag because CSP_M has no support for bags built-in.

6.1 Choice versus Concurrency

Consider the processes

$$\begin{aligned} P &= a \rightarrow b \rightarrow STOP \sqcap b \rightarrow a \rightarrow STOP \\ Q &= a \rightarrow STOP \mid_{\emptyset} b \rightarrow STOP. \end{aligned}$$

Evidently $P = Q$, although the latter offers concurrency of a and b not allowed by the former. That is revealed using our technique as follows.

By the definition of T ,

$$\begin{aligned} T(P) &= s.a \rightarrow e.a \rightarrow s.b \rightarrow e.b \rightarrow STOP \sqcap s.b \rightarrow e.b \rightarrow s.a \rightarrow e.a \rightarrow STOP \\ T(Q) &= s.a \rightarrow e.a \rightarrow STOP \mid_{\emptyset} s.b \rightarrow e.b \rightarrow STOP. \end{aligned}$$

To use FDR we define

$$SPEC = \sqcap x : \{\llbracket \rrbracket, \llbracket a \rrbracket, \llbracket b \rrbracket\} \bullet co.x \rightarrow SPEC.$$

Now $Ext(P) = P$ and $Ext(Q) = Q$ but using Ext_1 (see the explanation of definition [2])

$$SPEC \sqsubseteq_{\mathcal{T}} Ext_1(P) \setminus \{a, b\}$$

while

$$SPEC \not\sqsubseteq_{\mathcal{T}} Ext_1(Q) \setminus \{a, b\}.$$

The trace generated by FDR leading to the violation of the second refinement relation, namely

$$\langle co.\llbracket \rrbracket, co.\llbracket b \rrbracket, co.\llbracket a, b \rrbracket \rangle,$$

reveals that the events a and b might occur simultaneously in Q but not in P .

The same result can be obtained using the LTL model checking capabilities of ProB. We introduce the fresh event $conc_a_b$ and define

$$F(P) = P[co.\llbracket a, b \rrbracket \leftarrow conc_a_b]$$

to check if

$$\phi = G \text{ not } [conc_a_b]$$

holds on $F(Ext_1(P))$ or $F(Ext_1(Q))$. Expectedly ProB shows

$$\begin{aligned} F(Ext_1(P)) &\models \phi \\ F(Ext_1(Q)) &\not\models \phi. \end{aligned}$$

6.2 One-Place Buffer

A specification of a one-place buffer is

$$COPY = left?x \rightarrow right!x \rightarrow COPY.$$

The implementation (*SYSTEM*) presented in [3] is proved to be equivalent (in the failures-divergences model) to that specification. Since it uses parallel composition, it is of interest to check whether or not there are actually any concurrent events in the implementation; there might be concurrent τ events at least. There are no concurrent events in the implementation if the bag never grows beyond size one. This property can be built into the controller by modifying it as follows.

$$\begin{aligned} C_2(X) = \#X < 2 \ \& (s?x \rightarrow C_2(X \uplus \llbracket x \rrbracket)) \\ & \square sh?x \rightarrow C_2(X \uplus \llbracket x \rrbracket) \\ & \square e?x \rightarrow C_2(X \uplus \llbracket x \rrbracket) \\ & \square eh?x \rightarrow C_2(X \uplus \llbracket x \rrbracket) \\ & \square term \rightarrow SKIP) \end{aligned}$$

Checking $Hr((T(SYSTEM); term \rightarrow SKIP) \mid_{S_{con}} C_2(\llbracket \square \rrbracket)) =_{\mathcal{T}} COPY$ with FDR proves that there are no concurrent events in the implementation.

6.3 Dining Philosophers

The well-known example of the dining philosophers reveals a disadvantage of our approach: poor performance within FDR. Indeed the size of the transition system of the controller grows rapidly. In this example, N is the number of philosophers. The number of events available is $3N + 2N^2$. The set of all lists containing event labels and with maximum length n has $\sum_{i=0}^n (3N + 2N^2)^i$ elements, which is 20,440 for $N = 3$ and $n = 3$ or 551,881 for $N = 3$ and $n = 4$. Reducing these sets to create specifications like the one presented in the first example does not solve this problem. The set of all lists with maximum length n and at most one renamed *eat* event in it contains 19,756 elements for $N = 3$ and $n = 3$ and 517,420 in the case of $N = 3$ and $n = 4$. So checking the assertion that the *eat* event never occurs concurrently with another *eat* event with $N = 3$ takes about 90 minutes.

7 Related and Further Work

The work reported here relates particularly to other approaches to ‘non interleaving’ or ‘truly concurrent’ semantics of process algebra. In particular, it has been influenced by the work of Kwiatkowska and Phillips [7] and Taubner and Vogler [12]. In a sense, our approach combines the two by simulating the concurrency relation developed in the former, while maintaining the concurrent events in a structure that generalizes the steps defined in the latter.

In [7], Kwiatkowska and Phillips have proposed a ‘failures with divergence and conflict’ semantics for CSP. The semantics consists of a set of triples (F, D, C) , where F is a failure set, D a divergence set and C is a conflict relation on Σ^* , $C \subseteq \Sigma^* \times \Sigma^*$, defined by induction on the syntax of CSP. Furthermore, they distinguish *possible* conflict from *guaranteed* conflict. Analogously, they define a concurrency relation co . Our definitions 2 and 1 (and proposition 3 in particular) relate to their notions of possible concurrency and guaranteed conflict-freedom. In contrast to their work, rather than define a new semantics of CSP we have used the traces semantics \mathcal{T} to simulate ‘truly concurrent CSP’. In our approach, concurrency is encountered whenever the controller’s bag X grows beyond one element with frequency one. In particular, a single event is concurrent with itself if and only if there is a trace t such that its cardinality in the bag is greater than one after t . In contrast, each trace is by definition concurrent with itself in the semantics given by Kwiatkowska and Phillips.

Another important difference with that work is in refinement of concurrency information. In their semantics, only the refinement

$$a \rightarrow b \rightarrow STOP \sqsupseteq b \rightarrow a \rightarrow STOP \sqsubseteq a \rightarrow STOP \mid_{\emptyset} b \rightarrow STOP$$

holds but not conversely. Using the modified controller C_1 to incorporate concurrency information in the traces of a process yields the opposite refinement relation, as shown in example 6.1. Thus, in our approach, a process refines another with a higher level of concurrency.

In [12], Taubner and Vogler present a non-interleaving semantics of CSP based on the notion of ‘step’. In their semantics, a step is a finite bag of events from $\Sigma \cup \{\checkmark\}$. Traces and failures are lifted from sequences of events to sequences of steps, and refusals are defined over sets of steps. The empty step is called the null-step, and refusal of the null-step corresponds to divergence. A non-divergent process may never refuse the null-step. Their semantics generalizes the interleaving semantics of CSP in the sense that the special case of singleton steps is exactly the interleaving semantics.

That approach, like ours, realizes possible concurrency in the sense that whenever a step is possible, all of its sub-steps are also possible. One distinguishing feature of their semantics is that it lacks the commonly used τ event to model hidden actions. While the authors present this as a theoretical advantage because they succeed in establishing the common CSP laws in their semantics, it might be considered a disadvantage from a practical perspective. For that reason, our approach aims to detect any concurrent events, whether they are visible or hidden. The code related to an externally visible event a is likely to interfere with the code of a hidden a that is executed concurrently. Therefore, our controller registers even externally invisible events. Compared with their semantics, ours appears more verbose because it records not only the steps but also the creation of the steps (filling the bag).

On the level of traces our approach can be regarded as generalizing theirs: the controller can be modified to ensure that it refuses new s events after an e event until its bag is empty; that yields traces such that the state of the bag before

the first e after each non-empty sequence of s events are exactly the steps in their semantics. The present approach, like the step-failures semantics, can be used to optimize applications at runtime by predicting maximal parallelism. It also takes into account duration of user-defined functions related to events.

The language considered here overlooks several useful derived operators, like *chaining*, *interleaving*, *interrupt*, and *linked parallel* (a generalization of chaining), which are subject to future work. Of those, *interrupt* is especially interesting, because the interpretation of an event heavily influences its transformation by T . The issue is whether or not a single event x that is split into two events $s.x, e.x$ may be aborted. If a single event can be aborted, the event can no longer be interpreted as an action or function that transforms some state into a well-defined successor state. Furthermore, it would violate the e -non-refusability property (2). If a single event cannot be aborted, the transformed interrupt must take care of unfinished events, which renders the transformation reasonably complicated.

It is planned to realize *interleaving* according to the deparallelize operator proposed by Taubner and Vogler in [12]: $P \parallel Q$ corresponds to $P \bowtie_A Q$ where $A \subseteq \Sigma$ and \bowtie_A is an operator ensuring $\forall x, y : A \cdot (x, y) \notin \text{conc}(P \bowtie_A Q)$.

Another interesting issue is the extraction of concurrency in different ways. The method described here uses FDR to perform this by modifying the controller and checking whether or not the modified version violates the equality or querying fresh non-conflicting events. Unfortunately, FDR does not perform very well on such assertions because the bags grow rapidly and the functional part of CSP_M does not provide the most efficient way to model a process for analysis with FDR. One possible solution is to implement a specialized procedure that works on the original process P , to compute $\text{conc}(P)$. This procedure could be an extension of FDR's procedure to compute the traces of P that performs the transformation as well as synchronization with the controller internally before computing the traces.

It might be worthwhile to try the approach with the CSP prover [5], in view of the performance penalty caused by the controller process (see 6.3).

8 Conclusion

In this paper a construction has been given to replace each process with an equivalent version explicitly realising the possibility of concurrency. The controller C synchronizes with the transformed process. It maintains a bag X whose contents represent the events of the original process that are possibly concurrent after the trace that has lead to the current state. Thus it can be used in various ways to query concurrency information of a process.

In Example 6.1 the controller process C_1 of Definition 2 was used to emit the recorded concurrency information using a fresh event. That example has also demonstrated application of our approach with ProB.

The concurrency information can also be exploited using guards. A very simple application has been shown in the one-place-buffer example 6.2. Limiting the bag

to size 1 does not violate the equality. Thus, as expected, there are no concurrent events in the one-place buffer process.

Perhaps the most important aspect of our approach is that it allows reuse of existing CSP tools such as FDR [9], ProB [8] and the CSP prover [5], because it exploits semantics of CSP already implemented by those tools.

Acknowledgement

The first author acknowledges support from Deutsche Forschungsgemeinschaft (DFG) under the VATES project, grant number JA 379/17-1. The second author acknowledges support from the Macao Science and Technology Development Fund under the PEARL project, grant number 041/2007/A3.

References

1. Fischer, C.: CSP-OZ: A combination of Object-Z and CSP. In: FMOODS 1997: International Workshop on Formal Methods for Open Object-Based Distributed Systems, pp. 423–438. Chapman and Hall, Boca Raton (1997)
2. Gardner, W.B.: Bridging CSP and C++ with Selective Formalism and Executable Specifications. In: MEMOCODE 2003: International Conference on Formal Methods and Models for Co-Design, pp. 237–245. IEEE Computer Society, Los Alamitos (2003)
3. Goldsmith, M., Roscoe, B., Armstrong, P.: Failures-Divergence Refinement - FDR2 User Manual (2005), http://www.fsel.com/fdr2_manual.html
4. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall International, Englewood Cliffs (1985)
5. Isobe, Y., Roggenbach, M.: A Generic Theorem Prover of CSP Refinement. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 108–123. Springer, Heidelberg (2005)
6. Kleine, M., Sanders, J. W.: Simulating truly concurrent CSP. Technical Report 434, UNU-IIST, P.O. Box 3058, Macau (June 2010)
7. Kwiatkowska, M., Phillips, I.: Possible and Guaranteed Concurrency in CSP. In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 220–235. Springer, Heidelberg (1995)
8. Leuschel, M., Fontaine, M.: Probing the Depths of CSP-M: A new FDR-compliant Validation Tool. In: Liu, S., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 278–297. Springer, Heidelberg (2008)
9. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (2005)
10. Scattergood, B.: The Semantics and Implementation of Machine-readable CSP. PhD thesis, University of Oxford (1998)
11. Sun, J., Liu, Y., Dong, J.S.: Model Checking CSP Revisited: Introducing a Process Analysis Toolkit. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 307–322. Springer, Heidelberg (2008)
12. Taubner, D., Vogler, W.: Step failures semantics and a complete proof system. Acta Inf. 27(2), 125–156 (1989)

Appendix: Proof of Theorem 1

The proof proceeds by induction on the construction of P . Two cases are of special interest: Parallel composition because it is the only case that uses lemma 2 and external choice because, as explained in Section 3, hiding of initial events of the external choice's operands disallows distributivity of hiding over external choice in general. See the 6 for the whole proof.

Parallel Composition:

$$\begin{aligned}
 & Ext(P \mid_A Q) \\
 = & && \text{Lemma 1 and } T(P \mid_A Q) \\
 & Hr(T(P) \mid_{\{s.x, e.x \mid x \in A\}} T(Q)) \\
 = & && \text{Lemma 2} \\
 & Hr(T(P) \mid_{\{s.x \mid x \in A\}} T(Q)) \\
 = & && \text{distributivity of hiding and renaming over } \mid_{\{s.x \mid x \in A\}} \\
 & Hr(T(P)) \mid_A Hr(T(Q)) \\
 = & && \text{Lemma 1} \\
 & Ext(P) \mid_A Ext(Q) \\
 = & && \text{induction hypothesis} \\
 & P \mid_A Q.
 \end{aligned}$$

External Choice:

$$\begin{aligned}
 & Ext(P \square Q) \\
 = & && \text{Lemma 1 and } T(P \square Q) \\
 & Hr(T(P) \square T(Q)) \\
 = & && \text{definition of } Hr \\
 & (T(P) \square T(Q)) \setminus H_{Hr}[s.x \leftarrow x \mid x \in \Sigma_P \cup \Sigma_Q] \\
 =_{\mathcal{T}} & && \text{distributivity of hiding over } \square \text{ in } \mathcal{T} \\
 & (T(P) \setminus H_{Hr} \square T(Q) \setminus H_{Hr})[s.x \leftarrow x \mid x \in \Sigma_P \cup \Sigma_Q] \\
 = & && \text{distributivity of renaming over } \square \\
 & T(P) \setminus H_{Hr}[s.x \leftarrow x \mid x \in \Sigma_P] \square T(Q) \setminus H_{Hr}[s.x \leftarrow x \mid x \in \Sigma_Q] \\
 = & && \text{definition of } Hr \text{ and Lemma 1} \\
 & Ext(P) \square Ext(Q) \\
 = & && \text{induction hypothesis} \\
 & P \square Q.
 \end{aligned}$$

Statistical Verification of Probabilistic Properties with Unbounded Until

Håkan L.S. Younes¹, Edmund M. Clarke², and Paolo Zuliani²

¹ Google Inc

² Computer Science Department, Carnegie Mellon University, USA

Abstract. We consider statistical (sampling-based) solution methods for verifying probabilistic properties with unbounded until. Statistical solution methods for probabilistic verification use sample execution trajectories for a system to verify properties with some level of confidence. The main challenge with properties that are expressed using unbounded until is to ensure termination in the face of potentially infinite sample execution trajectories. We describe two alternative solution methods, each one with its own merits. The first method relies on reachability analysis, and is suitable primarily for large Markov chains where reachability analysis can be performed efficiently using symbolic data structures, but for which numerical probability computations are expensive. The second method employs a termination probability and weighted sampling. This method does not rely on any specific structure of the model, but error control is more challenging. We show how the choice of termination probability—when applied to Markov chains—is tied to the subdominant eigenvalue of the transition probability matrix, which relates it to iterative numerical solution techniques for the same problem.

1 Introduction

Probabilistic model checking deals with verification of stochastic systems, such as a queuing system with random arrivals and departures. Temporal stochastic logics, e.g., PCTL [12] and CSL [1], exist for expressing properties of such stochastic systems. Our focus is on *time-unbounded* properties of stochastic systems. For a queuing system, for instance, an interesting property could be: “the probability is at most 0.1 that the queue eventually becomes full.” In PCTL, such properties are expressed using the formula $\mathcal{P}_{\leq 0.1}[\top \mathcal{U} \text{full}]$.

We present two statistical algorithms for solving such model-checking problems that are based on *unbiased* sampling. Sampling is said to be unbiased if the expectation of the sample distribution is the same as the expectation of the true distribution. The use of unbiased sampling distinguishes our methods from most recent efforts to devise sampling-based algorithms for time-unbounded properties, which are based on *biased* sampling [2][18][27][3] (see Sect. 4).

Statistical algorithms for probabilistic model checking use discrete-event simulation to generate sample trajectories, and verify some temporal formula over each generated trajectory. This is combined with hypothesis testing or statistical

estimation to verify probabilistic properties [26,18]. The challenge for statistical algorithms with time-unbounded properties is to determine the truth-value of $\Phi \mathcal{U} \Psi$ without generating infinite sample trajectories.

The first method (see Sect.3.1) combines reachability analysis with statistical sampling. This approach has been used in the past for program analysis [20], and more recently for model checking [27] using biased sampling. The algorithm ensured termination for any finite-state homogeneous discrete-time Markov chain, although it is potentially applicable for any model for which we can perform reachability. The use of reachability analysis requires that we construct the full model, so it may seem counter to the appeal of statistical methods, which usually avoid model construction. The real cost in probabilistic model checking, however, lies in the numerical computation of probabilities, which we replace with sampling. We show in Sect.5 that the combination of reachability analysis and statistical sampling scales better with the size of the model than standard numerical solution methods. As a result, we can verify time-unbounded properties for larger models than possible with existing numerical algorithms. By using unbiased sampling, we can also make strong guarantees regarding error bounds. Other sampling-based methods, as well as iterative numerical solution methods, do not give the same strong guarantees as they depend on heuristics for bounding sample trajectory lengths or number of iterations.

The second method (see Sect.3.2) is based on a Monte Carlo method devised in the 1940s by John von Neumann and Stanislaw Ulam for computing the inverse of a matrix [10]. This method uses a termination probability p_T that is applied in each state along a trajectory to ensure finite sample trajectories. To account for the change in sample distribution, we weigh satisfying trajectories more heavily the longer the trajectory is. This way, we obtain an unbiased estimator of the probability that $\Phi \mathcal{U} \Psi$ holds over the set of trajectories that start in some state s . The second method does not rely on reachability analysis, so it has minimal memory requirements. It generally requires a larger number of sample trajectories to achieve the same precision as the first method, so it can be slower than the first method when reachability analysis is fast. It also suffers from the same problem as iterative numerical solution methods in that accuracy can be hard to guarantee. Still, the second method is potentially applicable for a much larger class of models.

We limit our attention to discrete-time Markov chains. The results extend trivially to continuous-time Markov chains and semi-Markov processes, as verification of time-unbounded properties for such models is done on an embedded discrete-time Markov chain.

2 Probabilistic Model Checking

This section describes discrete-time Markov chains (without nondeterminism), which is the class of models that we consider for probabilistic model checking. We present a temporal stochastic logic (a subset of PCTL) and discuss realistic error control for statistical model-checking algorithms.

2.1 Stochastic Processes and Discrete-Time Markov Chains

The terminology introduced here follows that of Stewart [22]. A *stochastic process* with state space S and time domain T is a family of random variables $\mathcal{X} = \{X_t \mid t \in T\}$. A random variable $X_t \in \mathcal{X}$ represents the outcome of observing the state of the stochastic process at time t .

A *discrete-time Markov process* is a stochastic process where T is the non-negative integers, \mathbb{Z}^* , and

$$\Pr[X_{n+1} = s_{n+1} \mid X_0 = s_0, \dots, X_n = s_n] = \Pr[X_{n+1} = s_{n+1} \mid X_n = s_n] \quad (1)$$

holds for all $n \in \mathbb{Z}^*$ and $s_i \in S$. If the state space is discrete as well, then we refer to the process as a *discrete-time Markov chain*. We will limit our attention to discrete-time Markov chains. The techniques we present later on can be generalized to other types of stochastic processes, but it is beyond the scope of this paper.

Let $p_{ij}(n) = \Pr[X_{n+1} = j \mid X_n = i]$, which denotes the probability of transitioning from state i at time n to state j at time $n + 1$. We call $p_{ij}(n)$, for all i and j in S and all $n \in \mathbb{Z}^*$, the transition probabilities of the discrete-time Markov chain. We have $p_{ij}(n) \in [0, 1]$ and, for all $i \in S$, $\sum_{j \in S} p_{ij}(n) = 1$. If, in addition to (I), we have $p_{ij}(n) = p_{ij}(m)$ for all n and m in \mathbb{Z}^* , then the discrete-time Markov chain is called *homogeneous*. In a homogeneous Markov chain, transition probabilities are independent of time. The transition probabilities of a finite-state homogeneous discrete-time Markov chain can be represented by a single $|S| \times |S|$ transition probability matrix \mathbf{P} . For notational convenience, we will use \mathbf{P} to represent the collection of transition probabilities, $p_{ij}(n)$, for nonhomogeneous Markov chains as well.

The evolution of a discrete-time Markov chain over time is captured by a *trajectory*. The trajectory of such a system is a sequence of states $\sigma = s_0 \rightarrow s_1 \rightarrow \dots$, with $s_i \in S$. We denote by $\sigma[i]$ the i th state, s_i , along the trajectory σ , and the finite prefix of length n of σ is denoted $\sigma \uparrow n$.

Let $Path(s)$ denote the set of trajectories with initial state s . Following Hansson and Jonsson [12], we define a probability measure μ on the set $Path(s)$, for each $s \in S$. The measure μ is defined on the probability space $\langle \Omega, \mathcal{F}_\Omega \rangle$, where $\Omega = Path(s)$, and \mathcal{F}_Ω is a σ -algebra generated by sets $\{\sigma \in Path(s) \mid \sigma \uparrow n = s \rightarrow s_1 \rightarrow \dots \rightarrow s_n\}$ of trajectories with common finite prefix of length n . The measure μ can then be defined uniquely by induction on the length of the common prefix as follows:

$$\mu(\{\sigma \in Path(s) \mid \sigma \uparrow 0 = s\}) = 1 \quad (2)$$

$$\begin{aligned} \mu(\{\sigma \in Path(s) \mid \sigma \uparrow n = s \rightarrow s_1 \rightarrow \dots \rightarrow s_n\}) = \\ \mu(\{\sigma \in Path(s) \mid \sigma \uparrow (n-1) = s \rightarrow \dots \rightarrow s_{n-1}\}) \cdot p_{s_{n-1}s_n}(n-1) \end{aligned} \quad (3)$$

2.2 Temporal Stochastic Logic

We use the Probabilistic Computation Tree Logic (PCTL [12]) to specify properties of discrete-time Markov chains. We describe only a subset of PCTL that

includes unbounded until, since that is the focus of this paper. The techniques described later in the paper can of course be combined with the techniques developed by Younes and Simmons [26] to handle a more expressive logic. The logic permits nested probabilistic operators, although we do not discuss such formulae here. Younes and Simmons [26] have already shown how to deal with nested probabilistic formulae without ties to any specific type of path formula. We could also replace PCTL with probabilistic LTL, which would avoid nested probabilistic operators altogether. The solution methods of this paper can be adapted for probabilistic LTL, but we do not consider that here.

Let AP be a fixed, finite set of atomic propositions. We assume a *labeled* discrete-time Markov chain $\mathcal{M} = \langle S, \mathbf{P}, L \rangle$. S , and \mathbf{P} as above, with the addition of a labeling function $L: S \rightarrow 2^{AP}$. $L(s)$ is the set of atomic propositions $a \in AP$ that are valid in s . PCTL formulae (for the relevant subset that we consider) are of the form

$$\Phi ::= a \mid \neg\Phi \mid \Phi \wedge \Phi \mid \mathcal{P}_{\bowtie\theta}[\Phi \mathcal{U} \Psi] ,$$

where $\theta \in [0, 1]$ and $\bowtie \in \{\leq, \geq\}$. Additional PCTL formulae can be derived in the usual way. For example, $\perp \equiv a \wedge \neg a$ for some $a \in AP$, $\top \equiv \neg\perp$, $\Phi \vee \Psi \equiv \neg(\neg\Phi \wedge \neg\Psi)$, $\Phi \rightarrow \Psi \equiv \neg\Phi \vee \Psi$, and $\mathcal{P}_{<\theta}[\varphi] \equiv \neg\mathcal{P}_{\geq\theta}[\varphi]$.

The standard logic operators have their usual meaning. $\mathcal{P}_{\bowtie\theta}[\varphi]$ asserts that the probability measure over the set of trajectories satisfying the path formula φ is related to θ according to \bowtie . Path formulae are constructed using the temporal path operator \mathcal{U} (“until”). The path formula $\Phi \mathcal{U} \Psi$ asserts that Ψ becomes true at some time $t \geq 0$ while Φ holds in all states prior to t . We can define mutually inductive satisfaction relations for PCTL state and path formulae as follows:

$$\begin{aligned} \mathcal{M}, s \models a & \quad \text{if } a \in L(s) \\ \mathcal{M}, s \models \neg\Phi & \quad \text{if } \mathcal{M}, s \not\models \Phi \\ \mathcal{M}, s \models \Phi \wedge \Psi & \quad \text{if } (\mathcal{M}, s \models \Phi) \wedge (\mathcal{M}, s \models \Psi) \\ \mathcal{M}, s \models \mathcal{P}_{\bowtie\theta}[\varphi] & \quad \text{if } \mu(\{\sigma \in Path(s) \mid \mathcal{M}, \sigma \models \varphi\}) \bowtie \theta \end{aligned}$$

$$\mathcal{M}, \sigma \models \Phi \mathcal{U} \Psi \quad \text{if } \exists i. ((\mathcal{M}, \sigma[i] \models \Psi) \wedge \forall j < i. (\mathcal{M}, \sigma[j] \models \Phi))$$

The fact that $\{\sigma \in Path(s) \mid \mathcal{M}, \sigma \models \varphi\}$ is measurable can be verified from the probability-space construction in Sect. 2.1 (cf. [1]), which makes the semantics for PCTL well-defined.

2.3 Error Control

Statistical solution methods cannot achieve the exact precision for probabilistic PCTL formulae that is required by the semantics given above. Following Younes and Simmons [25], we relax the semantics of PCTL by introducing an indifference region of half-width δ centered around any probability thresholds. The purpose is to quantify the error that we are willing to accept by using sampling and simulation in place of exact computations of probability measures.

Consider the model-checking problem $\mathcal{M}, s \models \mathcal{P}_{\bowtie\theta}[\varphi]$, and let p be the probability measure for the set of trajectories that start in s and satisfy φ . If $|p-\theta| < \delta$, then the truth value of $\mathcal{P}_{\bowtie\theta}[\varphi]$ is undetermined (“too close to call”) under the relaxed semantics; otherwise, it is the same as for PCTL.

Formally, given $\delta > 0$, we define two relations: \approx_{\top}^{δ} (approximate satisfaction) and \approx_{\perp}^{δ} (approximate “unsatisfaction”). The definitions of \approx_{\top}^{δ} and \approx_{\perp}^{δ} coincide with \models and $\not\models$, respectively, except for probabilistic formulae where we instead have:

$$\begin{aligned} \mathcal{M}, s \approx_{\top}^{\delta} \mathcal{P}_{\geq\theta}[\varphi] & \quad \text{if } \mu(\{\sigma \in Path(s) \mid \mathcal{M}, \sigma \approx_{\top}^{\delta} \varphi\}) \geq \theta + \delta \\ \mathcal{M}, s \approx_{\perp}^{\delta} \mathcal{P}_{\geq\theta}[\varphi] & \quad \text{if } \mu(\{\sigma \in Path(s) \mid \mathcal{M}, \sigma \approx_{\perp}^{\delta} \varphi\}) \leq \theta - \delta \\ \mathcal{M}, s \approx_{\top}^{\delta} \mathcal{P}_{\leq\theta}[\varphi] & \quad \text{if } \mu(\{\sigma \in Path(s) \mid \mathcal{M}, \sigma \approx_{\top}^{\delta} \varphi\}) \leq \theta - \delta \\ \mathcal{M}, s \approx_{\perp}^{\delta} \mathcal{P}_{\leq\theta}[\varphi] & \quad \text{if } \mu(\{\sigma \in Path(s) \mid \mathcal{M}, \sigma \approx_{\perp}^{\delta} \varphi\}) \geq \theta + \delta \end{aligned}$$

Let $\mathcal{M}, s \underline{accept}_{\mathcal{A}} \Phi$ represent the fact that Φ is accepted as true in state s of \mathcal{M} by a model-checking algorithm \mathcal{A} , and $\mathcal{M}, s \underline{reject}_{\mathcal{A}} \Phi$ that Φ is rejected as false in state s of \mathcal{M} by \mathcal{A} . The solution methods we present aim to guarantee the following error bounds:

$$\Pr[\mathcal{M}, s \underline{reject}_{\mathcal{A}} \Phi] \leq \alpha \quad \text{if } \mathcal{M}, s \approx_{\top}^{\delta} \Phi \quad (4)$$

$$\Pr[\mathcal{M}, s \underline{accept}_{\mathcal{A}} \Phi] \leq \beta \quad \text{if } \mathcal{M}, s \approx_{\perp}^{\delta} \Phi \quad (5)$$

The parameters α and β allow a user to control the probability of false negatives and false positives, respectively. For example, consider the formula $\mathcal{P}_{\geq 0.5}[\Phi \mathcal{U} \Psi]$ and let p denote the probability measure of trajectories that start in some state s and satisfy $\Phi \mathcal{U} \Psi$. Let $\delta = 0.01$. The statistical model-checking algorithms in this paper aim to guarantee that we reject the formula as false with probability at most α if $p \geq 0.5 + \delta = 0.51$, and that we accept the formula as true with probability at most β if $p \leq 0.5 - \delta = 0.49$. The three parameters α , β , and δ determine the precision of the model-checking algorithm. It is up to the user to set these to his or her satisfaction, with the understanding that higher precision will result in longer model-checking times.

3 Sampling-Based Verification of Unbounded Until

This section presents two methods for verifying probabilistic properties with unbounded until, based on statistical sampling. For the model-checking problem $\mathcal{M}, s \models \mathcal{P}_{\bowtie\theta}[\Phi \mathcal{U} \Psi]$, define the random variable $X : Path(s) \rightarrow \{0, 1\}$ as follows:

$$X(\sigma) = \begin{cases} 1 & \text{if } \mathcal{M}, \sigma \models \Phi \mathcal{U} \Psi \\ 0 & \text{if } \mathcal{M}, \sigma \not\models \Phi \mathcal{U} \Psi \end{cases} . \quad (6)$$

X represents a *Bernoulli trial* (i.e., outcomes are limited to 0 and 1). The expectation of X is

$$E[X] = \mu(\{\sigma \in Path(s) \mid \mathcal{M}, \sigma \models \Phi \mathcal{U} \Psi\}) . \quad (7)$$

Hence, $\mathcal{M}, s \models \mathcal{P}_{\geq \theta}[\Phi \mathcal{U} \Psi]$ has a positive answer if and only if $E[X] \asymp \theta$.

If we could sample observations of X , then we could use statistical hypothesis testing or estimation to verify $\mathcal{P}_{\geq \theta}[\Phi \mathcal{U} \Psi]$. To sample an observation of X , we would sample a trajectory from $Path(s)$ (e.g., using discrete-event simulation) and verify $\Phi \mathcal{U} \Psi$ over the sample trajectory. A single sample trajectory would be extended incrementally until we reach a state that satisfies either Ψ (positive observation; the path formula holds over the sample trajectory) or $\neg\Phi$ (negative observation; the path formula does not hold over the sample trajectory). The problem with this naive approach is that we are not guaranteed to ever reach a state that satisfies either Ψ or $\neg\Phi$. It works well for probabilistic properties with *time-bounded* until, as demonstrated by Younes and Simmons [26], because we can stop extending a sample trajectory if the time bound is exceeded. This ensures termination (with probability 1) provided that the model is time divergent. For unbounded until, however, there is no finite time bound to stop us from going on indefinitely, so we can no longer guarantee termination.

Consider, for example, the model in Fig. 1(a), with a single state satisfying some state formula Ψ . Assume that we want to verify $\mathcal{M}, s_0 \models \mathcal{P}_{\geq 0.15}[\top \mathcal{U} \Psi]$ (i.e., the probability of eventually reaching the state satisfying Ψ is at least 0.15 if we start in state s_0 at time 0). Note that the state satisfying Ψ is shown as absorbing—i.e., it has no outgoing transitions. This is to reflect that sample-trajectory generation ends in that state. For all other states, the outgoing transition probabilities sum to 1. Any trajectory that starts in s_0 and does not satisfy $\top \mathcal{U} \Psi$ is infinite. For this simple model, the probability measure of the set of satisfying trajectories that start in s_0 at time 0 can be computed as:

$$\mu(\{\sigma \in Path(s_0) : \mathcal{M}, \sigma \models \top \mathcal{U} \Psi\}) = 0.1 \cdot \sum_{i=0}^{\infty} 0.4^i = \frac{1}{6}. \quad (8)$$

Thus the stated model-checking problem has a positive answer, but a naive sampling-based approach does not work due to the positive probability ($\frac{5}{6}$) for the set of infinite trajectories.

Next, we describe two sampling-based solution methods that aim to avoid infinite sample trajectories.

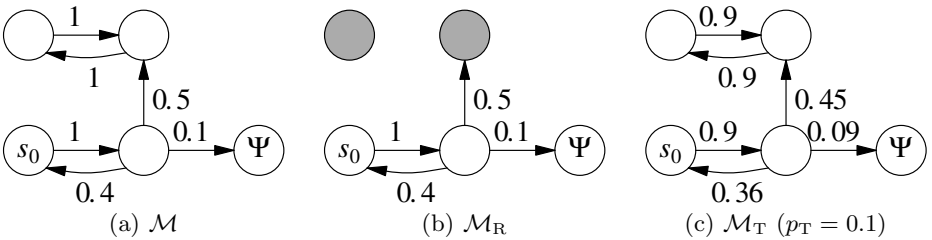


Fig. 1. Three variations of a simple discrete-time Markov chain

3.1 Sampling-Based Method with Reachability Analysis

The first method uses reachability analysis to avoid infinite sample trajectories. For an unbounded until formula $\Phi \mathcal{U} \Psi$ to hold over a single sample trajectory σ for model \mathcal{M} , it is necessary (although not sufficient) that $\mathcal{M}, \sigma[i] \models \Psi$ for some $i \geq 0$. If after the generation of a finite prefix $\sigma \uparrow n$ it becomes evident that $\neg\Psi$ invariably holds along all possible extensions of $\sigma \uparrow n$, then we can determine that $\Phi \mathcal{U} \Psi$ does not hold over σ without generating a complete (possibly infinite) sample trajectory.

This condition for early termination can be expressed formally as the *non-probabilistic* CTL [6] formula $AG \neg\Psi$, or equivalently $\neg EF \Psi$. Hence, if we first verify $EF \Psi$ for a model \mathcal{M} , which amounts to reachability analysis, then we can terminate the generation of any sample trajectory entering a state of \mathcal{M} that does not satisfy $EF \Psi$. This pre-processing step requires that we construct the full model, so it may seem counter to the appeal of sampling-based methods, which usually avoid model construction. We show, however, in Sect. 5 that this approach that combines symbolic reachability analysis and statistical sampling can work very well in practice. It scales better with the size of the model than numerical solution methods, which enables us to verify time-unbounded properties for larger models.

Let \mathcal{M}_R be the model we get by removing all outgoing transitions from all states in \mathcal{M} that do not satisfy $EF \Psi$. We can now define the Bernoulli random variable $X_R: Path_R(s) \rightarrow \{0, 1\}$ as follows:

$$X_R(\sigma) = \begin{cases} 1 & \text{if } \mathcal{M}_R, \sigma \models \Phi \mathcal{U} \Psi \\ 0 & \text{if } \mathcal{M}_R, \sigma \not\models \Phi \mathcal{U} \Psi \end{cases} . \quad (9)$$

Theorem 1. *Let X be the random variable defined in (6) and let X_R be the random variable defined in (9). The expectation of X_R is the same as the expectation of X : $E[X_R] = E[X]$.*

This theorem is a standard result in Markov chain theory (see, e.g., [2]), and a consequence of it is that we can use statistical hypothesis testing with observations of X_R instead of X to verify $\mathcal{P}_{\geq \theta}[\Phi \mathcal{U} \Psi]$ in \mathcal{M} . The benefit of using observations of X_R instead of X is that certain trajectories that are infinite in \mathcal{M} have been made finite in \mathcal{M}_R . Since X_R still represents a Bernoulli trial, the exact same techniques as for formulae with time-bounded until (described in detail by Younes and Simmons [26]) can be used to verify formulae with unbounded until for model \mathcal{M}_R , satisfying conditions (4) and (5).

We illustrate this solution method on the discrete-time Markov chain in Fig. 1. The original model is shown in Fig. 1(a). After performing reachability analysis, we obtain the model in Fig. 1(b). The gray states have been made absorbing because they do not satisfy $EF \Psi$. Trajectories generated from the modified model will almost surely (with probability 1) eventually terminate.

We assume here that reachability analysis can be performed efficiently on the model \mathcal{M} . For Markov chains, we can ignore the actual values of transition

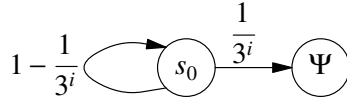


Fig. 2. A nonhomogeneous discrete-time Markov chain

probabilities and use BDD-based symbolic model checking [4]. Other models may require more advanced techniques (see, e.g., [13]). A discussion of concrete techniques for reachability analysis is beyond the scope of this paper. Clarke et al. [6] cover this topic in great depth.

For some *nonhomogeneous* Markov chains, termination may not be guaranteed (with probability 1) even after states have been made absorbing based on reachability analysis. Consider the nonhomogeneous Markov chain in Fig. 2, where i represents time. While $EF\Psi$ holds in s_0 , the probability measure of trajectories that start in s_0 and never terminate is approximately 0.56. Hence, if we applied the reachability-based approach to this model, more than half of the sample trajectories would never terminate.

This example shows that the reachability approach is not applicable to all Markov chains. The following theorem, however, identifies a large class of models for which this approach is applicable. This theorem, too, is standard in Markov chain theory (see, e.g., [2]).

Theorem 2. *If \mathcal{M} is a finite-state homogeneous discrete-time Markov chain, then the probability measure is zero for the set of infinite trajectories of \mathcal{M}_R .*

3.2 Sampling-Based Method with Termination Probability

The first solution method cannot be used if reachability analysis is ineffective as exemplified by the model in Fig. 2. In the case of infinite-state systems, reachability analysis may not even be feasible.

To ensure finite trajectories without relying on reachability analysis, we can introduce a *termination probability* $p_T < 1$, which is used as follows. Start with the stochastic discrete-event system \mathcal{M} . Let \mathcal{M}_T be the system we get if before each transition out of a non-absorbing state in \mathcal{M} we terminate execution prematurely with probability p_T . Concretely, for a discrete-time Markov chain with transition probabilities $p_{ij}(n)$, we construct a new discrete-time Markov chain with transition probabilities $(1 - p_T) \cdot p_{ij}(n)$. Figure 1(c) shows the result of this transformation on the model in Fig. 1(a) using termination probability $p_T = 0.1$ (we note later on that there are limitations on the choice of p_T —0.1 is not an admissible choice for all models). Each transition probability in \mathcal{M} is multiplied by $1 - p_T$ to obtain the corresponding transition probability in \mathcal{M}_T . For example, 0.5 becomes $(1 - 0.1) \cdot 0.5 = 0.45$. The outgoing transition probabilities now sum to $1 - p_T$ for all non-absorbing states. In reality, of course, we never construct the new Markov chain. Instead we just take the termination probability into account when we generate sample trajectories. At each state, we terminate the generation of the trajectory prematurely with probability p_T .

Let $|\sigma|$ denote the number of state transitions along the trajectory σ . Define the random variable $X_T: \text{Path}_T(s) \rightarrow [0, \infty)$ as follows:

$$X_T(\sigma) = \begin{cases} (1 - p_T)^{-|\sigma|} & \text{if } \mathcal{M}_T, \sigma \models \Phi \mathcal{U} \Psi \\ 0 & \text{if } \mathcal{M}_T, \sigma \not\models \Phi \mathcal{U} \Psi \end{cases}. \quad (10)$$

Trajectories that satisfy $\Phi \mathcal{U} \Psi$ are finite as they must terminate in a Ψ -satisfying state, so (10) is well-defined. Note the *negative* exponent, which means that the weight of a satisfying trajectory grows exponentially in the length of the trajectory. This construction is due to von Neumann and Ulam (see [10,11]) as a way to compute the inverse of a matrix by the Monte Carlo method.

Theorem 3. *Let X be the random variable defined in (6) and let X_T be the random variable defined in (10). The expectation of X_T is the same as the expectation of X : $E[X_T] = E[X]$.*

We can thus use observations of X_T instead of X to solve the model-checking problem $\mathcal{M}, s \models \mathcal{P}_{>\theta}[\Phi \mathcal{U} \Psi]$. Unlike X_R , which represents a Bernoulli trial, the distribution of X_T is unknown. Therefore we cannot use the same efficient hypothesis-testing techniques as before. However, because $E[X_T] = E[X]$ we can use an estimation-based approach. If we can obtain an estimate \tilde{p} of $E[X_T]$, then we can decide $\mathcal{P}_{>\theta}[\Phi \mathcal{U} \Psi]$ by comparing \tilde{p} to the threshold θ . While model checking using \mathcal{M}_T requires more expensive sampling techniques than \mathcal{M}_R , we can show that it is more generally applicable.

Theorem 4. *The probability measure is zero for the set of infinite trajectories of \mathcal{M}_T .*

Note that Theorem 4 does not depend on any property of \mathcal{M} , so we are guaranteed (with probability one) finite sample trajectories for any model. For example, Theorem 4 applies to the nonhomogeneous Markov chain in Fig. 2, as well as to any infinite-state Markov process and even general discrete-event systems.

It remains to find a way to estimate $E[X_T]$. Chow and Robbins [5] provide such a method. Their method is a sequential procedure for computing a fixed-width confidence interval for a random variable with unknown but *finite* variance. We can use their procedure to obtain a confidence interval for $E[X_T]$ of width 2δ centered around a point estimate \tilde{p} with coverage probability at least $1 - \alpha$. Let x_i be the i th observation of X_T and let \bar{x}_n be the arithmetic mean of the first n observations. Furthermore, let a_1, a_2, \dots be a sequence of positive constants such that $\lim_{n \rightarrow \infty} a_n = \Phi^{-1}(1 - \frac{\alpha}{2})$, where Φ^{-1} is the inverse standard normal cumulative distribution function (in practice, we can choose a_n to be the $1 - \frac{\alpha}{2}$ quantile of the t -distribution with $n - 1$ degrees of freedom). The stopping rule for the sequential procedure is then given by [5, Eqn. 3]:

$$N = \inf \left\{ n \geq 1 : \frac{1}{n} + \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x}_n)^2 \leq \frac{\delta^2 n}{a_n^2} \right\} \quad (11)$$

We can now use $\tilde{p} = \bar{x}_N$ as a point estimate for $E[X_T]$, and accept $\mathcal{P}_{\bowtie\theta}[\Phi \mathcal{U} \Psi]$ as true if and only if $\tilde{p} \bowtie \theta$. As shown by Younes [25], this gives us a model-checking procedure that satisfies conditions (4) and (5) with $\beta = \alpha$.

It should be noted that the procedure of Chow and Robbins provides *asymptotic* guarantees only, meaning that the coverage probability of the confidence interval is guaranteed to be $1 - \alpha$ in the limit as δ approaches 0. In practice, the coverage probability can be somewhat less than $1 - \alpha$ for any selected δ , no matter how small, as has been shown for the normal distribution [9]. On the other hand, empirical coverage tends to be greater than $1 - \alpha$ for Bernoulli random variables. Further empirical studies are needed to determine the empirical coverage for the type of random variables we have here, which are neither normal nor Bernoulli, but this is beyond the scope of our paper.

A prerequisite for using the procedure of Chow and Robbins is that X_T has finite variance. This restriction effectively limits our choice of the termination probability p_T . For finite-state homogeneous Markov chains, we have the following theoretical result:

Theorem 5. *Let \mathbf{P} be the transition probability matrix for \mathcal{M} (the original model). X_T has finite variance iff $p_T < 1 - \rho(\mathbf{P})$, where $\rho(\mathbf{P})$ is the subdominant (second-largest) eigenvalue of \mathbf{P} .*

In practice, computing the subdominant eigenvalue of a stochastic matrix is no easier than solving the model-checking problem at hand, so choosing the right value for p_T is not trivial. In Sect. 5, we apply the algorithm to *parametric models* [6] and compute $\rho(\mathbf{P})$ for small models to find a p_T that is likely to give finite variance for larger variations of the same basic model. It is important to note, however, that numerical iterative solution methods suffer from the exact same problem as discussed in the next section.

4 Related Work

To verify the formula $\mathcal{P}_{\bowtie\theta}[\Phi \mathcal{U} \Psi]$ in some state s , we can first compute the probability measure p of the set of trajectories that start in s and satisfy $\Phi \mathcal{U} \Psi$, and then compare p to θ . A numerical computation of p for any state of a Markov-chain model amounts to the solution of a set of linear equations specified as follows (cf. [1]). Let \mathbf{P} be the transition probability matrix of the Markov chain and let \mathbf{P}' equal \mathbf{P} , with the exception that states satisfying $\neg\Phi \vee \Psi$ have been made absorbing. Furthermore, let \mathbf{v} be a binary column vector with a 1 in each row corresponding to a state that satisfies Ψ . Then \mathbf{p} is the solution to

$$\mathbf{p} = \mathbf{P}' \cdot \mathbf{p} + \mathbf{v} . \quad (12)$$

The equation system in (12) can be written as $(\mathbf{I} - \mathbf{P}') \cdot \mathbf{p} = \mathbf{v}$ and solved using Gaussian elimination, which is guaranteed to be polynomial in the size of the state space. This approach is memory intensive, however, and also suffers from numerical instability. For these reasons, iterative solution methods, such as Jacobi and Gauss-Seidel [22], are typically preferred. The leading tool for

probabilistic model checking, PRISM [17], relies on iterative methods to verify properties with unbounded until. Each iteration involves a matrix–vector multiplication, which in the worst case is $O(n^2)$, but often $O(n)$ (for sparse models), where n is the size of the state space. This dependence on the size of the state space make numerical solution methods impractical for very large models, in which case sampling-based solution become an attractive alternative.

The number of iterations (k) required to achieve some numerical precision ϵ is related to the subdominant eigenvalue (ρ) of \mathbf{P}' as follows [22, p.156]: $k = \frac{\log \epsilon}{\log \rho}$. Since computing eigenvalues is no easier than solving the model-checking problem, heuristics must be employed to bound the number of iterations, but this means that no formal correctness guarantees can be made. This is similar to the situation for the second sampling-based method we described. The reachability-based sampling approach, in contrast, does not suffer from this weakness as the precision of the result is independent of any property of the model.

John von Neumann and Stanislaw Ulam, as early as the 1940s, devised a Monte Carlo method for solving systems of linear equations of the type in (12). It is this algorithm, first published by Forsythe and Leibler [10], that we use in the second of our solution methods. It should come as no surprise that both the iterative numerical solution methods, and the Monte Carlo approach that uses a termination probability, have a dependency on the subdominant eigenvalue to provide some prescribed precision. The method of von Neumann and Ulam is essentially a Monte Carlo version of a numerical iterative algorithm.

Sampling-based solution methods for time-unbounded formulae have received some attention recently [21,18,317], but these authors appear unaware of the method devised by von Neumann and Ulam.

Sen et al. [21] propose a solution method that on the surface looks similar to our second approach (the one based on the method by von Neumann and Ulam). They use a termination probability p_T in the same way as we to ensure terminating sample trajectories. Instead of using weighted sampling with the random variable X_T , however, they use the following Bernoulli random variable:

$$Y_T(\sigma) = \begin{cases} 1 & \text{if } \mathcal{M}_T, \sigma \models \Phi \mathcal{U} \Psi \\ 0 & \text{if } \mathcal{M}_T, \sigma \not\models \Phi \mathcal{U} \Psi \end{cases} . \quad (13)$$

The problem with Y_T is that its expectation does not match that of the random variable X . Sen et al. recognize this problem and provide a bound on the expectation $E[X]$ expressed in terms of $E[Y_T]$. The bound depends on the size of the model, however, and is too loose to be useful in practice.

In other work [18,273], it is proposed to use the results from verifying time-bounded properties to obtain a probability estimate for unbounded properties. These methods essentially boil down to estimating the expected value of the following random variable:

$$Z_k(\sigma) = \begin{cases} 1 & \text{if } \mathcal{M}, \sigma \uparrow k \models \Phi \mathcal{U} \Psi \\ 0 & \text{if } \mathcal{M}, \sigma \uparrow k \not\models \Phi \mathcal{U} \Psi \end{cases} . \quad (14)$$

As with Y_T , Z_k does not have the same expectation of X , although we do have $\lim_{k \rightarrow \infty} Z_k = X$. The expected value for Z_k is estimated for a series of increasing values for k until some convergence criterion is met. Lassaigne and Peyronnet [18] relate the choice of k to the subdominant eigenvalue, but as with the other theoretical results mentioned earlier that involve the subdominant eigenvalue, this result does not help to choose k in practice. Ensuring a certain accuracy becomes hard because each successive iteration with a different value for k is subject to error, and different ad-hoc termination criteria are proposed by the various authors. In the solution methods presented in this paper, we avoid this iterative estimation approach by always setting up experiments that preserve the expected value of the quantity of interest.

Rabih et al. [7] present a very different simulation-based approach to verifying unbounded until properties. They develop an algorithm based on perfect simulation. The approach is interesting, but impractical unless the model is monotone. The authors do not discuss how to determine monotonicity for a model, or whether the widely-used PRISM benchmarks satisfy this property, so it is hard to assess the general applicability of their method.

The termination-probability approach has been used for rare-event simulation (see, e.g., [19]). Improvements to the basic algorithm from the simulation community would be valuable for model checking as well.

5 Empirical Evaluation

We use two continuous-time Markov-chain models as the basis for our empirical evaluation, with rather different characteristics. Note that model checking of unbounded until for continuous-time Markov-chains reduces to model checking for the embedded discrete-time Markov chain [1]. Thus, a continuous-time Markov chain presents no additional challenge for the algorithms described in Sect. 3.

5.1 Modified Polling System

The first model is a variation of an n -station symmetric polling system, described by Ibe and Trivedi [16]. In the original polling-system model, each station has a single-message buffer and the stations are attended to by a single server in cyclic order. When attending to a station, the server checks for a message in the buffer of the station by polling the station, and goes on to serve the station if there is a message. The polling and service times are exponentially distributed with rates $\gamma = 200$ and $\mu = 1$, respectively. Furthermore, each station has an inter-arrival time for messages that is exponentially distributed with rate $\lambda = 1/n$.

Here, we consider a modified version of the polling-system model, where polling stations can fail and stop accepting messages. The failure rate of a station is $\kappa = \lambda/3$. After a station has failed, it can never be served again. This way, infinite sample trajectories become a possibility for the property we consider.

We verify the following property: the probability is at least 0.4 that station 1 is served before station 2. Let $s \in \{1, \dots, n\}$ be the station currently attended

to by the server, and let $a \in \{0, 1\}$ represent the activity of the server (0 for polling and 1 for serving). The property can then be expressed as the formula $\mathcal{P}_{\geq 0.4}[\neg(s=2 \wedge a=1) \mathcal{U} s=1 \wedge a=1]$. We verify this formula in the state where the server is about to attend to station 1 ($s=1$ and $a=0$) and all buffers are empty. If both station 1 and station 2 fail before they are served, $\neg(s=2 \wedge a=1)$ will remain true and $s=1 \wedge a=1$ false indefinitely, and a sample trajectory for the given model-checking problem may never terminate.

We have implemented the two sampling-based algorithms described in this paper in YMER [24]. We compare the sampling-based approaches with numerical iterative algorithms implemented in PRISM [17]. For the experiments, we used $\alpha = \beta = 0.01$ and $\delta = 0.005$ for the sampling-based algorithms, and $\epsilon = 0.005$ (absolute error) for the numerical algorithms. These choices are of course somewhat arbitrary. Younes and Simmons [26] provide a thorough investigation of how these parameter choices affect performance for sampling-based algorithms. For example, using $\alpha = \beta = 10^{-8}$ increases verification time by about a factor 10. For the reachability approach, we tried both Wald’s SPRT [23] (sequential hypothesis testing) and estimation based on the Hoeffding bound [15] that gives a fixed sample size of $N = \lceil \frac{1}{2\delta^2} \log \frac{2}{\alpha} \rceil$ (105,967 for our choice of parameters). For the approach based on termination probability we used Chow and Robbins’ sequential estimation procedure mentioned earlier and $p_T = 10^{-4}$ (this choice was made after computing the subdominant eigenvalue for the transition matrices of models with up to 12 stations). Finally, for the numerical approaches, we used the hybrid engine in PRISM, trying both Jacobi and (backwards) Gauss-Seidel.

Figure 3 plots the verification time as a function of state-space size for the modified polling system, when verifying the stated property using different algorithms. For the sampling-based approaches, the graph shows the average time over 20 trials. The state-space size for a model with n stations is $4n \cdot 3^{n-1}$. A model with 24 stations, for example, has close to 10^{13} states.

The sampling-based approaches scale well as the state-space size grows. Reachability combined with the SPRT does best, partly because the underlying probability is close to 0.5, while the bound in the formula is 0.4, so a decision can be reached with an average sample size of about 1,200. The estimation-based approaches require much larger sample sizes. The termination-based approach beats the reachability-based approach when the latter is used with a sample size derived from the Hoeffding bound.

The numerical algorithms are much faster for small state spaces, but performance deteriorates quickly for larger state spaces. They also use more memory than the sampling-based approaches. For a model with 16 stations (about 1 billion states), PRISM caused serious thrashing on a computer equipped with 8 GB of RAM. The reachability-based approach has a much more modest memory growth, and nothing suggests that it could not handle models much larger than what we have tested here. The termination-based approach uses the least amount of memory, as it does not require reachability analysis, and does not show any noticeable growth as models get bigger. The sampling-based approaches are trivially parallelizable, so we could get a speedup on multi-core architectures.

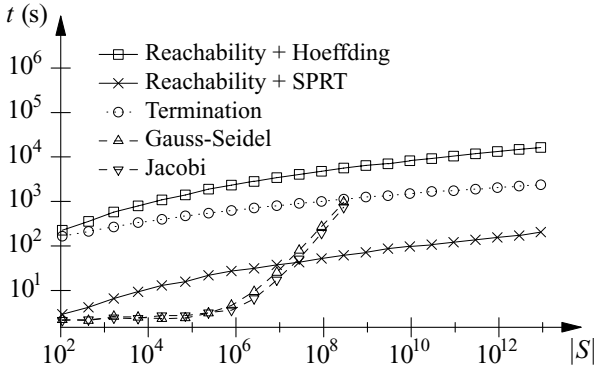


Fig. 3. Verification time as a function of state-space size for modified polling system

5.2 Tandem Queuing Network

The second model is a tandem queuing network due to Hermanns et al. [14]. The network consists of two serially connected queues, each with capacity n . Messages arrive at the first queue, get routed to the second queue, and eventually leave the system from the second queue. The inter-arrival time for messages at the first queue is exponentially distributed with rate $\lambda = 4n$. The processing time at the second queue is exponentially distributed with rate $\kappa = 4$. The size of the state space for this model is $O(n^2)$.

We verify that the probability is at most 0.03 that the second queue becomes full before the first queue: $\mathcal{P}_{\leq 0.03}[-full_1 \cup full_2]$. We use the same experimental setup as for the first model, except for the choice of p_T . Instead of a fixed value, we use $p_T = \frac{1}{n+2}$ for a model with queues of size n . We do so because the subdominant eigenvalue for this model more quickly approaches 1 as n grows. Figure 4 plots the verification time as a function of state-space size for the

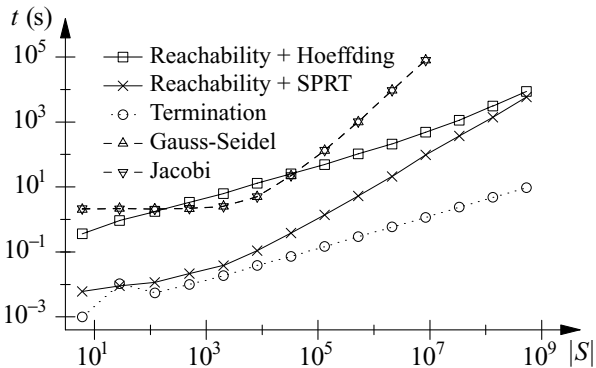


Fig. 4. Verification time as a function of state-space size for tandem queuing network

tandem queuing network, using the same algorithms as before. Again, the results for sampling-based approaches are averages over 20 trials.

For this model, the sampling-based algorithm that uses termination probability comes out on top. The reason is that reachability analysis is much more expensive for this model. The difference between Hoeffding and SPRT gets smaller for larger state spaces, as the time needed to perform reachability analysis starts to dominate the difference in sample size (446 for SPRT; 105,967 for Hoeffding).

6 Discussion

We have presented two sampling-based algorithms for probabilistic model checking of time-unbounded properties. Both solution methods are based on unbiased estimators. This avoids the convergence issues that haunt existing sampling-based algorithms, which all use biased estimators. The first method, especially, is valuable as it provides correctness guarantees that are independent of any model parameters. The second method has the same weakness as popular iterative numerical solution methods, in that accuracy cannot be guaranteed fully without knowledge of the subdominant eigenvalue. Still, it allows us to analyze models far beyond the reach of methods that require model construction (as is the case for the first method, as well as numerical methods). Future work could focus on extending the theoretical results of this paper. In particular, conditions under which X_T has finite variance should be established for a more general class of systems. Techniques from the simulation community should be incorporated to reduce the variance of X_T , and the empirical coverage probability for sequential estimation should be established.

Acknowledgments. Edmund M. Clarke and Paolo Zuliani were supported in part by the GSRC under contract no. 1041377, National Science Foundation under award no. CNS0926181, no. CCF0541245, and no. CNS0931985, Semiconductor Research Corporation under contract no. 2005TJ1366, General Motors under contract no. GMCMUCRLNV301, Air Force under contract no. 18727S3, and the Office of Naval Research under award no. N000141010188.

References

1. Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.-P.: Model-checking algorithms for continuous-time Markov chains. *IEEE Transactions on Software Engineering* 29(6), 524–541 (2003)
2. Baier, C., Katoen, J.-P.: *Principles of Model Checking*. The MIT Press, Cambridge (2008)
3. Basu, S., Ghosh, A.P., He, R.: Approximate model checking of PCTL involving unbounded path properties. In: Breitman, K., Cavalcanti, A. (eds.) *ICFEM 2009*. LNCS, vol. 5885, pp. 326–346. Springer, Heidelberg (2009)

4. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. *Information and Computation* 98(2), 142–170 (1992)
5. Chow, Y.S., Robbins, H.: On the asymptotic theory of fixed-width sequential confidence intervals for the mean. *Annals of Mathematical Statistics* 36(2), 457–462 (1965)
6. Clarke, E.M., Grumberg, O., Peled, D.A.: *Model Checking*. The MIT Press, Cambridge (1999)
7. El Rabih, D., Pekergin, N.: Statistical model checking using perfect simulation. In: Liu, Z., Ravn, A.P. (eds.) *ATVA 2009*. LNCS, vol. 5799, pp. 120–134. Springer, Heidelberg (2009)
8. Etesami, K., Rajamani, S.K. (eds.): *CAV 2005*. LNCS, vol. 3576. Springer, Heidelberg (2005)
9. Fishman, G.S.: *Monte Carlo: Concepts, Algorithms, and Applications*. Springer, Heidelberg (1996)
10. Forsythe, G.E., Leibler, R.A.: Matrix inversion by a Monte Carlo method. *Mathematical Tables and Other Aids to Computation* 4(31), 127–129 (1950)
11. Hammersley, J.M., Handscomb, D.C.: Solution of linear operator equations. In: *Monte Carlo Methods*, ch. 7, pp. 85–96. Methuen & Co, New York (1964)
12. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. *Formal Aspects of Computing* 6(5), 512–535 (1994)
13. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. *Information and Computation* 111(2), 193–244 (1994)
14. Hermanns, H., Meyer-Kayser, J., Siegle, M.: Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains. In: *Proc. 3rd International Workshop on the Numerical Solution of Markov Chains*, pp. 188–207, Prensas Universitarias de Zaragoza (1999)
15. Hoeffding, W.: Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association* 58(301), 13–30 (1963)
16. Ibe, O.C., Trivedi, K.S.: Stochastic Petri net models of polling systems. *IEEE Journal on Selected Areas in Communications* 8(9), 1649–1657 (1990)
17. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic symbolic model checking with PRISM: A hybrid approach. *International Journal on Software Tools for Technology Transfer* 6(2), 128–142 (2004)
18. Lassaigne, R., Peyronnet, S.: Probabilistic verification and approximation. *Annals of Pure and Applied Logic* 152(1–3), 122–131 (2008)
19. L’Ecuyer, P., Demers, V., Tuffin, B.: Splitting for rare-event simulation. In: *Proc. 2006 Winter Simulation Conference*, pp. 137–148. IEEE, Los Alamitos (2006)
20. Monniaux, D.: An abstract monte-carlo method for the analysis of probabilistic programs. In: *Proc. 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 93–101. Association for Computing Machinery (2001)
21. Sen, K., Viswanathan, M., Agha, G.: On statistical model checking of stochastic systems. In: Etesami, K., Rajamani, S.K. (eds.) *CAV 2005*. LNCS, vol. 3576, pp. 266–280. Springer, Heidelberg (2005)
22. Stewart, W.J.: *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, Princeton (1994)
23. Wald, A.: Sequential tests of statistical hypotheses. *Annals of Mathematical Statistics* 16(2), 117–186 (1945)

24. Younes, H.L.S.: Ymer: A statistical model checker. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 429–433. Springer, Heidelberg (2005)
25. Younes, H.L.S.: Error control for probabilistic model checking. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 142–156. Springer, Heidelberg (2005)
26. Younes, H.L.S., Simmons, R.G.: Statistical probabilistic model checking with a focus on time-bounded properties. *Information and Computation* 204(9), 1368–1409 (2006)
27. Zapreev, I.S.: Model checking Markov chains: Techniques and tools. PhD thesis, University of Twente (2008)

Reasoning about Assignments in Recursive Data Structures

Alejandro Tamalet and Ken Madlener

Institute for Computing and Information Sciences (iCIS),
Radboud University Nijmegen, The Netherlands
`{a.tamalet,k.madlener}@cs.ru.nl`

Abstract. This paper presents a framework to reason about the effects of assignments in recursive data structures. We define an operational semantics for a core language based on Meyer’s ideas for a semantics for the object-oriented language Eiffel. A series of field accesses, e.g. $f_1 \bullet f_2 \bullet \dots \bullet f_n$, can be seen as a path on the heap. We provide rules that describe how these *multidot* expressions are affected by an assignment. Using multidot expressions to construct an abstraction of a list, we show the correctness of a list reversal algorithm. This approach does not require induction and the reasoning about the assignments is encapsulated in the mentioned rules. We also discuss how to use this approach when working with other data structures and how it compares to the inductive approach. The framework, rules and examples have been formalised and proven correct using the PVS proof assistant.

1 Introduction

In order to verify pointer programs that manipulate recursive data structures, one generally identifies the pointer structure embedded in the heap with an abstract model. A concrete instance is a mapping of a set of objects on the heap connected by a field such as `next` to an abstract list of objects. The mapping is called the *abstraction* and the abstract list is called the *abstract model*. An operation performed by the program on a pointer structure on the heap has a corresponding operation on the abstract model. For example, the operations performed by a list reversal algorithm have the combined effect that the abstract list is reversed at the end of the execution. The standard way to define data abstractions is by recursion on the structure of (the data type of) the abstract model.

Verification of pointer programs is a non-trivial task due to the possibility of aliasing. Modifying data through one name implicitly modifies the values associated to all aliased names. If two portions of the heap are disjoint, an assignment in one part of the heap does not affect the other; this is called *local reasoning*. Local reasoning is essential for scalability and several approaches to obtain it have been studied, see e.g. Separation Logic [14] and Region Logic [15].

When it is not known how the heap is partitioned or when working within a region that may contain aliases, we have to reason about how a change to

(a portion of) the heap affects the corresponding abstract model. This complements local reasoning. In this paper we focus on the effects of assignments to abstract models. We present our work in the setting of a core language, inspired by Meyer’s ideas for a semantics for the object-oriented language Eiffel [12].

Our framework allows us to express multidot field access expressions, multidot expressions for short, of the form $f_1 \cdot f_2 \cdot \dots \cdot f_n$. A multidot expression consisting of a series of `next`-fields describes a path from the head of a list to one of its elements. If we instantiate it with a series of `left` and `right`-fields we can describe the path from the root of a binary tree to any node or leaf. In general, a multidot expression describes a path on the heap where the elements are connected by field accesses.

The main contribution of this paper is to provide a set of rules that precisely describe the value of a multidot expression after an assignment, and to show how these rules can be applied for verification of programs that manipulate recursive data structures. The given rules are categorised into separation rules, where the assignment has no effect on the multidot expression, and interference rules, where the assignment does have effect on the multidot expression. We have applied these rules to show the correctness of an in-place list reversal algorithm by mapping each element of the list to a multidot expression. We also discuss how to apply the same principles to other recursive data structures and we make a comparison with the standard inductive approach. Our work has been completely carried out in the theorem prover PVS [13].

This paper is organised as follows. Section 2 gives a short introduction to PVS and introduces the notation. Section 3 defines the language we shall work with. In Section 4 we present the rules that describe the effects an assignment can have on a multidot expression and in Section 5 we apply these rules to prove the correctness of a list reversal algorithm and we discuss the applicability to other data structures. We compare the approach described in this paper with the standard inductive approach and we give pointers for future work in Section 6. Related work is discussed in Section 7 and conclusions are drawn in Section 8.

2 Preliminaries

PVS is based on higher-order logic with dependent types and predicate subtyping. Subtyping based on predicates makes type checking undecidable, so when PVS cannot infer the desired type itself, it will generate a proof obligation. Its intuitive syntax is reminiscent of functional languages like Haskell. For reasons of presentation, we slightly simplify the actual PVS syntax. We will briefly introduce the notation used in this paper.

Formulas are terms of type `bool`. We shall use the standard notation for connectives ($\wedge, \vee, \Rightarrow, \neg$), and for quantifiers (\forall, \exists). There is a conditional term **IF** φ **THEN** M **ELSE** N , for terms M, N of the same type.

Given the types $\sigma, \tau, \sigma_1, \dots, \sigma_n$, function types are written as $[\sigma \rightarrow \tau]$ and record types as $[\text{lab}_1 : \sigma_1, \dots, \text{lab}_n : \sigma_n]$. Given the record types ρ_1, \dots, ρ_m , labelled coproduct types are written as $\{\text{lab}_1(\rho_1), \dots, \text{lab}_m(\rho_m)\}$. Terms of

coproduct type can be constructed with $\mathbf{lab}_i(M)$, where $M : \rho_i$, and recognised with $\mathbf{lab}_i?$. Standard set comprehension notation can be used to define predicate subtypes. New types can be introduced via definitions, like

$$\mathbf{lift}[\sigma] : \mathbf{TYPE} = \{\mathbf{bottom}, \mathbf{up}(\mathbf{down} : \sigma)\}$$

(\mathbf{bottom} is the unit type and we omit its argument). The \mathbf{lift} type constructor adds a \mathbf{bottom} element to an arbitrary type σ given as a parameter, written in short as $\sigma_{\perp} \hat{=} \mathbf{lift}[\sigma]$.

Lists are defined as

$$\mathbf{list}[\sigma] : \mathbf{TYPE} = \{\mathbf{null}, \mathbf{cons}(\mathbf{car} : \sigma, \mathbf{cdr} : \mathbf{list})\}$$

There is an infix function $\mathbf{++}$ that appends two lists. It is overloaded so that when one of its arguments is of type σ , then this argument is converted to a list. The i th element of a list \mathbf{l} can be accessed using $\mathbf{nth}(\mathbf{l}, i)$.

3 The Model

This section describes an operational semantics of a core object-oriented language. The focus is on the features needed to understand the properties discussed in the next section, i.e., we do not model some typical object-oriented features like inheritance. The interested reader can find the full PVS formalisation at <http://cs.ru.nl/~tamalet>.

3.1 The Heap

In our model we consider all values to be an object or void. The set \mathbf{Object} is defined as an uninterpreted type that represents non-void objects. Instances of \mathbf{Object}_v have the possibility of being void:

$$\mathbf{Object}_v : \mathbf{TYPE} = \{\mathbf{obj}(\mathbf{obj} : \mathbf{Object}), \mathbf{void}\}$$

A basic approach to model the heap, due to Burstall [5] and more recently emphasised by Bornat [3], is to model it as a collection of functions of type $\mathbf{Object} \rightarrow \mathbf{Object}_v$, one for each class field (i.e. the component). This modelling encodes the fact that changing to what object a field points to does not affect other fields. This has the important consequence that whenever one field is updated, we do not need to propagate that update to the other fields. This is sometimes called the component-as-array model [749].

Our heap is a grouping of field functions, indexed by their field names:

$$\mathbf{Heap} : \mathbf{TYPE} = [\mathbf{Name} \rightarrow [\mathbf{Object} \rightarrow \mathbf{Object}_v]]$$

where \mathbf{Name} is a set representing the field names. Given a heap \mathbf{h} and a field name \mathbf{f} , $\mathbf{h}(\mathbf{f})$ is the corresponding field function. This indexing allows us to reason about field names, which is not possible when using a loose set of field functions as in the component-as-array model. There, the names of the fields are fixed by the names of the functions that model them. We use this to express meta-properties

about multidot field expressions in Section 4. The separation by syntax provided by the component-as-array model is lost in this model, because a field update is now an update on the heap function. With the meta-level properties presented in the rest of this paper, we obtain a reincarnation of separation by syntax.

The above definition of the heap highlights the relationship with the component-as-array model. However, defining the heap as a function of type $[\text{Object} \rightarrow [\text{Name} \rightarrow \text{Object}_v]]$ may seem more intuitive. In this definition we first fix an object and then we ask for a field name to obtain its value. As the functions are total (required by PVS), both definitions are in fact equivalent. This means that every field should be defined at every object. This is of course not realistic, however, accesses to undefined fields can be handled by a preliminary static analysis.

3.2 Expressions, Statements and Compositions

We model expressions, statements and their compositions following Meyer's ideas for a semantics for Eiffel [12]. A distinctive aspect of this approach is that expressions and statements are evaluated relative to an object, which is provided together with the heap as argument.

We deal with null-pointer dereferencing in language constructs, as opposed to avoiding it by type constraints. In our experience, the second approach leads to cumbersome specifications because the result of each expression and statement must be checked for definedness before composing them.

There are two syntactic categories: expressions (without side-effects) and statements:

$$\begin{aligned} \text{Expr} : \text{TYPE} &= \{e : [\text{Object}_{v\perp}, \text{Heap}_\perp \rightarrow \text{Object}_{v\perp}] \mid \\ &\quad \forall (o : \text{Object}_{v\perp}, h : \text{Heap}_\perp) : \\ &\quad \quad \text{bottom_or_void?}(o, h) \Rightarrow \text{bottom?}(e(o, h))\} \\ \\ \text{Stmt} : \text{TYPE} &= \{S : [\text{Object}_{v\perp}, \text{Heap}_\perp \rightarrow \text{Heap}_\perp] \mid \\ &\quad \forall (o : \text{Object}_{v\perp}, h : \text{Heap}_\perp) : \\ &\quad \quad \text{bottom_or_void?}(o, h) \Rightarrow \text{bottom?}(S(o, h))\} \end{aligned}$$

To define a semantics for Eiffel, Meyer works with partial functions [12]. In most theorem provers, including PVS, functions have to be total. For this reason we use lifted arguments, to represent undefinedness. The `bottom_or_void?(o,h)` predicate returns `true` if and only if `o` is undefined or void or `h` is undefined. By using predicate subtypes, we ensure that whenever an expression or statement is evaluated in void or in an undefined object or state, the result is undefined. This shifts checking for void or bottom from the specification to type correctness obligations that PVS generates automatically.

The expression `Current` (called `this` or `self` in some languages) returns the current object:

$$\begin{aligned} \text{Current} : \text{Expr} &= \lambda (o : \text{Object}_{v\perp}, h : \text{Heap}_\perp) : \\ &\quad \text{IF } \text{bottom_or_void?}(o, h) \text{ THEN } \text{bottom} \text{ ELSE } o \end{aligned}$$

The operators \bullet and $;$ compose expressions and statements. If x is an expression, S an statement and r is either of them, we define:

$$\begin{aligned} S ; r &= \lambda (o : \text{Object}_{v\perp}, h : \text{Heap}_{\perp}) : r(o, S(o, h)) \\ x \bullet r &= \lambda (o : \text{Object}_{v\perp}, h : \text{Heap}_{\perp}) : r(x(o, h), h) \end{aligned}$$

The normal uses are state compositions $S;T$ and field access $x \bullet y$. The overloading allows us also to write $S; x$, which returns the value of evaluating x after the statement S , and $x \bullet S$, which can be thought as a qualified call of S from x .

We define in PVS an automatic conversion that translates a name f into the expression $\lambda (o : \text{Object}_{v\perp}, h : \text{Heap}_{\perp}) : h(f)(o)$ whenever needed. This allows us to express a field access directly as $x \bullet f$. We also define a conversion that translates an $o : \text{Object}$ into $\text{obj}(o) : \text{Object}_v$, and one that translates an $o : \text{Object}_v$ into $\text{up}(o) : \text{Object}_{v\perp}$, to reduce the amount of syntax.

IF-statements are mapped to **IF**-expressions in the logic of PVS. For reasons of succinctness we omit the treatment of **WHILE**.

3.3 Assignments

At its core, an assignment is an update of a heap-function in a particular point (consisting of a field and an object):

$$\begin{aligned} \text{update}(f : \text{Name}, p : \text{Object}, h : \text{Heap}, q : \text{Object}_v) : \text{State} = \\ \lambda (g : \text{Name})(o : \text{Object}) : \\ \text{IF } p = o \wedge f = g \text{ THEN } q \text{ ELSE } h(g)(o) \end{aligned}$$

Our model forces us to explicitly deal with undefinedness due to dereferencing of void. The **update** operation is encapsulated in an operator $:=$ that assigns an object q to the field f of the object p in the heap h .¹

$$\begin{aligned} := (f : \text{Name}, q : \text{Object}_v) : \text{Stmt} = \\ \lambda (p : \text{Object}_{v\perp}, h : \text{Heap}_{\perp}) : \\ \text{IF } \text{bottom_or_void?}(p) \vee \text{bottom?}(h) \text{ THEN } \text{bottom} \\ \text{ELSE } \text{update}(f, \text{obj}(\text{down}(p)), \text{down}(h), q) \end{aligned}$$

If the assignment is made in an undefined state or tries to assign to void, the error is propagated. This is required by the definition of **Stmt**. We shall use the above variable names throughout the rest of this paper. The next step is to define local assignments $f := e$ and qualified assignments $e_1 \bullet f := e_2$. These definitions are not relevant for the development of this paper and we therefore omit them.

An assignment affects a field access if and only if the object where the field is evaluated is the one where the assignment was made and the field being accessed is the one that was assigned to. This is summarised in the following two basic separation and interference properties (both assume that o, h is not bottom or void):

Property 1. If $p \neq o$ or $f \neq g$, then $g(o, (f := q)(p, h)) = g(o, h)$.

¹ In the PVS formalisation we have called this function \leq , because $:=$ is reserved.

Property 2. If $p = o$ and $f = g$, then $g(o, (f := q)(p, h)) = q$.

The proofs of these two properties amount to expanding the definition of $:=$ and applying several case-splits. When the assignment is replaced with a qualified assignment $e_1 \bullet f := e_2$, then analogous properties hold, but $p = o$ is replaced by $e_1(o, h) = o$.

One has to explicitly apply properties [1](#) and [2](#) as proof steps to reason about the effect of an assignment in the presented semantics. The key condition is $p = o \wedge f = g$. The latter is a syntactical comparison and thus can be done automatically. However, most of the time comparison between objects cannot be discharged automatically, unless we have information about the layout of the heap, see Section [6](#).

4 The Effect of Assignments on Multidot Expressions

In this section we look at expressions of the form

$$(g_1 \bullet \dots \bullet g_n)(o, (f := q)(p, h)), \quad (1)$$

where the g_i and f are field names, o and p are $\text{Object}_{v\perp}$ and q is of type Object_v . Because undefinedness due to dereferencing void is not an essential part of the discussion, we shall omit it in the rest of the paper.

Properties [1](#) and [2](#) describe the result of a very simple multidot, namely one where n is equal to 1. There, the condition which determines the result is $p = o \wedge f = g$. In multidot field expressions of arbitrary length a similar condition determines the result, but now it must be taken into account that in the path from o to $(g_1 \bullet \dots \bullet g_n)(o, (f := q)(p, h))$, the field f of the object p can be traversed more than once if there is a loop. Thus we are interested in the set of indexes k such that:

$$p = (g_1 \bullet \dots \bullet g_{k-1})(o, h) \text{ and } f = g_k.$$

The properties we present in this section are categorised into *separation rules*, where the assignment has no effect on the multidot field expression, and *interference rules*, where the assignment does have effect on the multidot field expression. Moreover, we now have a choice to look at the heap h before the assignment, or at the heap $h' = (f := q)(p, h)$ after the assignment. For the separation properties this does not make a difference, but for the interference properties it does.

The properties we derive about multidot expressions in this section are at the meta-level. Although it is possible to use them to reason about a particular multidot in a program, the intended use is to reason about the effects of assignments on recursive data structures. Examples that demonstrate the application are given in Section [5](#).

To improve readability, the notation for multidot expressions differs from the actual syntax used in PVS. In the last subsection we show the concrete PVS formalisation of a property. We will use graphs representing a portion of the

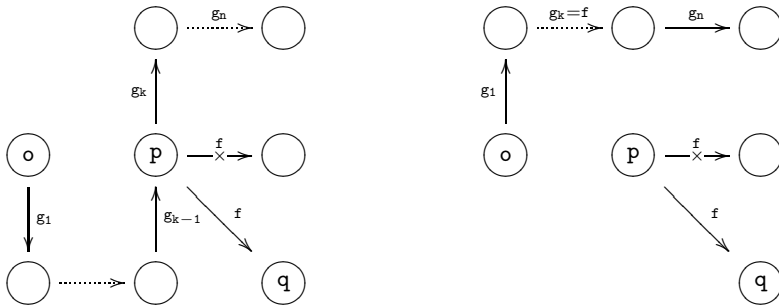
heap h' to show examples of the properties. In these graphs nodes are objects and edges are labelled with an attribute name. An edge \xrightarrow{f} from an object o to an object p means that $f(o, h') = p$. The edge removed by the heap update is depicted as $\xrightarrow{\cancel{f}}$.

4.1 Looking at the Heap Before the Assignment

The assignment in (II) may or may not modify the multidot field expression. Graphically, what matters is whether the edge that has changed belongs to path followed by the multidot field expression or not. A particular edge is determined by its object of origin and the field name. Hence, the condition that determines whether the assignment influences the multidot expression is whether or not the following set is empty:

$$K_{\text{pre}} \hat{=} \{k : \text{nat} \mid k < n \wedge p = (g_1 \cdot \dots \cdot g_{k-1})(o, h) \wedge f = g_k\}.$$

We start with the case where K_{pre} is empty, i.e., the edge changed by the assignment is not part of the multidot expression, as shown in Figure I.



(a) $p = (g_1 \cdot \dots \cdot g_{k-1})(o, h)$ but $f \neq g_k$. (b) $f = g_k$ but $p \neq (g_1 \cdot \dots \cdot g_{k-1})(o, h)$.

Fig. 1. Examples where K_{pre} is empty

As the edge that changed was not part of the multidot expression, the assignment does not have an effect on it.

Property 3. ($\text{empty_}K_{\text{pre}}$) If K_{pre} is empty, then

$$(g_1 \cdot \dots \cdot g_n)(o, h') = (g_1 \cdot \dots \cdot g_n)(o, h).$$

Now consider the case where K_{pre} is not empty. Figure II depicts an example with two indexes i and k in K_{pre} such that $k < i$. If there are several indexes in K_{pre} , it means that there are several loops starting at p . The assignment breaks the first edge in these loops. In the heap after the assignment, the edge that joins p with q is determined by the *least* element in K_{pre} .

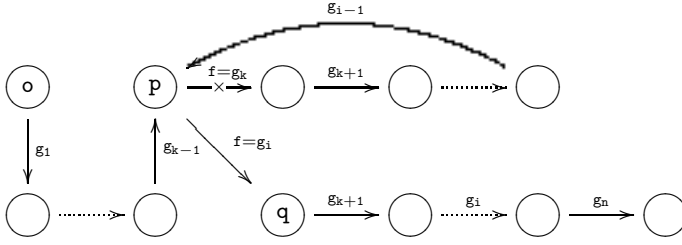


Fig. 2. Example with two indexes $k < i$ in K_{pre}

Property 4. (\min_K_{pre}) If $k = \min(K_{pre})$, then

$$(g_1 \cdot \dots \cdot g_n)(o, h') = (g_{k+1} \cdot \dots \cdot g_n)(q, h').$$

Since the assignment may also affect the path that goes from q to the final value, the right hand side must still be evaluated in h' .

4.2 Looking at the Heap After the Assignment

Instead of looking at when the multidot expression follows the edge that changed in h , we will now look at when it follows the new edge in h' . That is, we will look at the set:

$$K_{pos} \hat{=} \{k : \text{nat} \mid k < n \wedge p = (g_1 \cdot \dots \cdot g_{k-1})(o, h') \wedge f = g_k\}.$$

If the new edge is never traversed, the multidot expression does not change.

Property 5. ($\text{empty_}K_{pos}$) If K_{pos} is empty, then

$$(g_1 \cdot \dots \cdot g_n)(o, h') = (g_1 \cdot \dots \cdot g_n)(o, h).$$

Now assume that there is at least one index in K_{pos} . In Figure 3 we see an example with two such indexes i and k with $i < k$. In this case the result of the multidot expression can be described as either $(g_{k+1} \cdot \dots \cdot g_n)(q, h')$ or as $(g_{i+1} \cdot \dots \cdot g_k \cdot g_{k+1} \cdot \dots \cdot g_n)(q, h')$. If we take the greatest index in K_{pos} , we get the shortest path to the resulting value and since the rest of the edges are not affected by the assignment we can describe the result in terms of h . This is expressed in the following properties.

Property 6. ($\text{forall_}K_{pos}$) For all k in K_{pos} ,

$$(g_1 \cdot \dots \cdot g_n)(o, h') = (g_{k+1} \cdot \dots \cdot g_n)(q, h').$$

Property 7. ($\text{max_}K_{pos}$) If $k = \max(K_{pos})$, then

$$(g_1 \cdot \dots \cdot g_n)(o, h') = (g_{k+1} \cdot \dots \cdot g_n)(q, h).$$

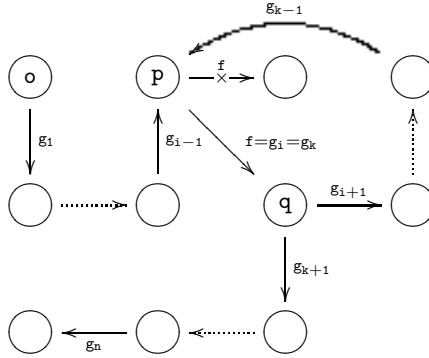


Fig. 3. Example with two indexes $i < k$ in K_{pos}

4.3 PVS Formalisation

Given a list of names fs , the dot composition of the corresponding attributes is formalised as

```

multidot(fs : list[Name]) : RECURSIVE Expr =
  IF null?(fs) THEN Current
  ELSIF null?(cdr(fs)) THEN car(fs)
  ELSE car(fs) . multidot(cdr(fs))
MEASURE length(fs)

```

Note that because $e \cdot \text{Current} = e$ does not hold when e evaluates to void, we cannot simply append `Current` at the end of the `multidot` expression.

As an example of the PVS formalisation, we show a property that combines `empty_K_pos` and `max_K_pos` in a property at the source code level. Since it is written as an equality between functions, it can be used as a rewrite rule.

```

multidot_after_assignment_pos : LEMMA
  ∀ (f : Name, gs : list[Name], x, e : Expr,
     o : Objectv⊥, h : Heap⊥) :
  ((x . f := e ; multidot(gs))(o, h) =
   LET h' = (x . f := e)(o, h),
    K_pos = λ (k: below[length(hs)]) :
      x(o, h) = multidot(take(gs, k))(o, h') ∧
      f = nth(gs, k) IN
   IF bottom?(h') THEN bottom
   ELSIF empty?(K_pos) THEN multidot(gs)(o, h)
   ELSE LET k = max(K_pos) IN
     IF k = length(gs) - 1 THEN e(o, h)
     ELSE (e . multidot(drop(gs, k+1)))(o, h)

```

This property describes in terms of h all the possible outcomes of `multidot(gs)` when evaluated in h' . If the assignment resulted in an error then the result is an

error. If K_{pos} is empty then the multidot expression is unchanged. Otherwise, let k be the greatest element in K_{pos} . The result is then as stated in \max_K_{pos} (with a shift of indexes due to lists starting at 0 in PVS). But again because $e \bullet \text{Current}$ is not equal to e when evaluated on void, we have to make a special case for when the multidot expression ends exactly at e . There is a similar lemma that combines $\text{empty_}K_{\text{pre}}$ and $\text{min_}K_{\text{pre}}$.

The intuitive way to prove these properties is by induction on gs . The intention is to reason about the last edge of the multidot expression and use the inductive hypothesis on the path that leads to it. The problem with this approach is that on the non-empty case we have to reason about a list of the form $\text{cons}(g, \text{gs})$. Therefore, we get to reason about the first edge, not the last one. To overcome this problem we defined a function multidot_rev that chains the arguments in the reverse order. Then we wrote lemmas that are adapted to work with the reversed list, and we proved them by induction on gs . Finally, the original lemmas were proven using their reversed counterpart by instantiating gs with $\text{reverse}(\text{gs})$.

5 Linearised Abstractions

In this section we look at examples of abstract models expressed in terms of multidot field expressions. We call this style of specifying *linearised*, because it is not by recursion on the structure of the abstract model. The properties derived in the previous section provide us a set of tools to reason about the effects of an assignment to a linearised abstraction.

5.1 Paths

The following definition abstracts a path embedded in the heap to a list l of `Object`s. The i th object in l is the object on the heap that can be accessed by requesting the first i fields describing the path.

$$\begin{aligned} & \text{Path}(\text{gs} : \text{list}[\text{Name}], l : \text{list}[\text{Object}]) \\ & \quad (\text{o} : \text{Object}_{\text{v}\perp}, \text{h} : \text{Heap}_{\perp}) : \text{bool} = \\ & \quad \text{length}(\text{gs}) + 1 = \text{length}(l) \wedge \\ & \quad \forall (i : \text{below}[\text{length}(l)]) : \\ & \quad \quad \text{multidot}(\text{take}(\text{gs}, i))(\text{o}, \text{h}) = \text{nth}(l, i) \end{aligned}$$

Due to the possibility of undefinedness, we define the abstractions as predicates about the heap and the current object rather than as functions because in PVS functions must be total.

With the use of the spatial separation lemmas for multidot expressions we can prove the following separation lemma for paths (recall that $\text{h}' = (\text{f} := \text{q})(\text{p}, \text{h})$):

Property 8. If for all $i < \text{length}(l)$ it holds that $\text{p} \neq \text{nth}(l, i)$ or $\text{f} \neq \text{nth}(\text{gs}, i)$, and $\neg \text{bottom}?(f(\text{p}, \text{h}))$, then

$$\text{Path}(\text{gs}, l)(\text{o}, \text{h}') = \text{Path}(\text{gs}, l)(\text{o}, \text{h}).$$

Thinking again in terms of graphs, this lemma says that if an edge outside the path is modified, then the path is not affected by the assignment. To give an idea of how the multidot rules are applied, we sketch the proof of this lemma.

Proof sketch. We are supposed to show that the `Path` predicates are logically equivalent. In expanded form, we have to show that the following predicates are equivalent:

$$\forall (i_1 : \text{below}[\text{length}(l)]) : (g_1 \bullet \dots \bullet g_{i_1})(o, h') = \text{nth}(l, i_1) \quad (2)$$

$$\forall (i_2 : \text{below}[\text{length}(l)]) : (g_1 \bullet \dots \bullet g_{i_2})(o, h) = \text{nth}(l, i_2) \quad (3)$$

To show that (2) implies (3), we instantiate i_1 with i_2 and we apply `empty_Kpos`. Then we have to show that `Kpos` is indeed empty. If this was not the case then there would be a k such that

$$p = (g_1 \bullet \dots \bullet g_{i_k})(o, h') = \text{nth}(l, k) \quad \text{and} \quad f = g_k,$$

which contradicts the assumption that p is not in l . For the converse direction, we apply `empty_Kpre` in an analogous way. \square

The interference property for paths describes how a path ending in p can be joined with a path beginning at q :

Property 9. If $p \notin l_0 \uparrow q \uparrow l_1$ and $c = \text{car}(l_0 \uparrow p)$, then

$$\text{Path}(gs_1 \uparrow f \uparrow gs_2, l_0 \uparrow p \uparrow q \uparrow l_1)(c, h') = \\ (\text{Path}(gs_1, l_0 \uparrow p)(c, h) \wedge \text{Path}(gs_2, q \uparrow l_1)(q, h))$$

The proof uses the multidot rules `empty_Kpos` and `max_Kpos` for the implication from left to right and it uses the rules `empty_Kpre` and `min_Kpre` from right to left. We omit the proof sketch of this property for reasons of space.

An important point about the proofs using linearised abstractions is that the induction is encapsulated in the rules about multidot expressions; to prove the above properties, we did not apply induction.

5.2 Example: Verification of an In-Place List Reversal Algorithm

The `Path` abstraction can be specialised by `Path(g, l)`, which instantiates the regular `Path` with a list of g -fields. By requiring the last node of `Path(next, l)` to point to void, we obtain an abstraction for lists on the heap:

```
List(l : list[Object])
  (o : Objectv, h : Heapl) : bool =
  Path(next, l)(o, h) ∧
  IF cons?(l) THEN void?(next(last(l), h))
  ELSE void?(o)
```

Note that $\text{List}(\text{null})(o, h)$ is true iff $\text{void?}(o)$ is true, i.e. an empty list is represented by void. Similar separation and interference properties as the ones for Path can be proved for List .

To prove the correctness of the annotated in-place list reversal algorithm listed in Figure 4, we use standard Hoare-style reasoning. The annotations have type $\text{Asrt} : [\text{Object}_{\perp}, \text{Heap}_{\perp} \rightarrow \text{bool}]$ and a Hoare-triple has the following meaning for $P, Q : \text{Asrt}$ and $S : \text{Stmt}$:

$$\{P\} S \{Q\} \hat{=} \forall (o : \text{Object}_{\perp}, h : \text{Heap}_{\perp}) : P(o, h) \Rightarrow Q(o, S(o, h))$$

As can be seen in Figure 4, the current object o and the updated heap $S(o, h)$ distribute over the connectives. So, the actual work to verify the correctness of the list reversal algorithm amounts to simplifying expressions of the form $(g \bullet \text{List}(l))(o, (e_1 \bullet f := e_2)(o, h))$. By expanding the definitions of dot and assignment, this can be brought into the form of $\text{List}(l)(o', (f := q)(p, h'))$, on which the separation and interference rules for the List abstraction can be applied.

```

{ λ(o, h) : ¬bottom_or_void?(o, h) ∧ a • List(As)(o, h) }
b := void;
WHILE (λ(o, h) : ¬void?(a(o, h))) DO
{ λ(o, h) : ¬bottom_or_void?(o, h) ∧
  ∃(as, bs : list[Object]) :
    (a • List(as))(o, h) ∧ (b • List(bs))(o, h) ∧
    disjoint?(as, bs) ∧ append(reverse(as), bs) = reverse(As) }
  tmp := a;
  a := a • next;
  tmp • next := b;
  b := tmp;
OD
{ λ(o, h) : ¬bottom_or_void?(o, h) ∧ (b • List(reverse(As)))(o, h) }

```

Fig. 4. In-place list reversal

5.3 Other Data Structures

The linearised specification approach exemplified in the previous two sections can also be applied to other recursive data structures. Consider for example binary trees that store a value in each node:

```
binary_tree[σ] : TYPE = {leaf, node(v : σ, l, r : binary_tree)}
```

It is straightforward to define a predicate

```
get_node(bt : binary_tree[σ], path : list[Name], v : σ) : bool
```

that says whether by traversing `bt` in the order specified by `path`, we arrive at `v`. Basically `get_node` maps each constructor application to the corresponding field name. We can now describe a binary tree on the heap by mapping each of its nodes to a multidot field access:

```
binary_tree_abstraction(bt : binary_tree[Object])
    (o : Objectv,⊥, h : Heap⊥) : bool =
  ∀ (x : Object, path : list[Name]) :
    get_node(bt, path, x) ⇒
      multidot(path)(o, h) = x
```

From the properties about multidot expressions presented in Section 4 one can obtain separation and interference lemmas for binary trees.

The same ideas can be applied to other tree-like structures. First make a linearised abstraction of the data structure: obtain the path from the root to each of its elements and use that path to describe the pointer structure in terms of multidot expressions. Then use the properties of Section 4 when reasoning about assignments. Data structures with loops can also be specified, e.g., a circular list is just a path that starts and ends in the same object.

6 Evaluation and Future Work

A natural way to define abstractions is by means of recursion on the structure of the abstract model. We single out the work by Mehta and Nipkow that uses this approach to verify several pointer programs [11]. The advantage of using induction is that it is a familiar general-purpose method that is integrated in the theorem prover. Much work has been devoted to automate proofs by induction, in particular to heuristics to instantiate the inductive hypothesis, e.g. *rippling* [4]. In the inductive approach one still has to reason about the effect of the assignments to the data structure, whereas using the rules given in Section 4 the focus is on when to apply each rule and in finding the extrema of the K-sets, which requires an instantiation.

Our experience is that both approaches require a comparable amount of proof work. However, there is still work to be done on investigating specialised version of the assignment rules and on the integration with the theorem prover as tactics. For example, if we know that there is no loop on a multidot expression, as is the case in tree-like structures, then we also know that the K-sets are either empty or have only one element. This eliminates the need to find the minima or the maxima.

Because both approaches lead to definitions that are essentially equivalent, the same properties hold. Hence, our approach can be seen as a complement rather than a replacement of inductive reasoning.

Reasoning about assignments ultimately reduces to reasoning about object equality. Therefore, this framework would benefit from knowledge about the layout of the memory. The separation rules are used to provide local reasoning, but they are not a primitive of the logic as the star conjunct is in Separation

Logic [14] (see also Section 7). Hubert and Marché [9] propose a static separation analysis and show how it can be integrated in the component-as-array modelling. They split the heap into regions that are inferred by the separation analysis and accordingly relabel the field names as a combination f_r of the old field name f and a region r . This could be integrated into our model, for example by redefining the heap as

$$\text{Heap} : \text{TYPE} = [\text{Region}, \text{Name} \rightarrow [\text{Object} \rightarrow \text{Object}_v]]$$

When it is inferred that two objects x and y lie in separate regions, the comparison between them can be avoided and the separation lemmas can be applied automatically.

7 Related Work

A first version of some of the rules presented in Section 4 first appeared in Tamalet’s Master’s thesis [16].

In the seminal work of Bornat [3] and also in the work by Meyer on a semantics for Eiffel [12], pointer structures on the heap are related with abstract models via repeated composition of field requests. This has been a source of inspiration for this paper. Bornat and Meyer both define a sequence closure operator that repeatedly requests a series of (the same) fields, yielding the list of objects that is traversed on the heap. This is essentially the same as our `Path` abstraction of Section 5.2. In this paper we have given a complete and formalised overview of the effects of assignments to arbitrary multidot field expressions. A treatment of the sequential operator in the context of Eiffel has been given in an unpublished work by Blanco and Castro [2], restricted to the case of lists.

A perhaps more natural way to define abstractions is by the use of recursion on the structure of the abstract model. Mehta and Nipkow [11] used this approach to verify the correctness of several pointer programs. We have compared the inductive approach and the linearised approach in Section 6.

Hoare and Jifeng [8] introduce a framework for the formulation of assertions about objects and pointers based on trace model of graphs and process algebra. They use a graphical notation very similar to the one used in this paper. However, their model uses graph transformations to describe the changes to the state whereas we use an operational semantics.

Our rules about an assignment followed by a multidot are meta-level properties of the language. To enable this meta-level reasoning we introduced a function `multidot` that maps a list of `Names` to a suitable expression, which is essentially a deep embedding of multidot expressions. The rules about multidot expressions are a reflection of the properties 1 and 2. For an instructive paper on reflection with examples in PVS we refer to [18].

Local Reasoning

Local reasoning is the key to scalability in formal verification of programs. The way the heap is modelled in our framework is based on the component-as-array

modelling idea of Burstall [5]. Refinements of this modelling have been used as the core of weakest pre-condition calculus-based tools such as Krakatoa for the verification of Java programs, and Caduceus for the verification of C programs [7,10]. A separation analysis tailored to integration with the component-as-array modelling has been proposed by Hubert and Marché [9]. Future work on the integration of this analysis with our work has been discussed in Section 6.

A well-studied approach to obtain local reasoning is that of Separation Logic, proposed by Reynolds [14], which can be seen as a radical refinement of Burstall's idea. In Separation Logic disjointness of portions of the heap is made explicit in the logic. Its frame rule allows one to reason about just the relevant portion of the heap that a piece of code manipulates and later augment it with the rest of the heap. So far, no concrete case studies on industrial software make use of Separation Logic, but there is ongoing research on its automation, see e.g. [6,11]. An implementation of [1] has been developed inside the theorem prover HOL by Tuerk [17].

A related line of research is Region Logic, whose goal it is to preserve the local reasoning of Separation Logic, but without using non-standard semantics of Hoare-triples. See [15] for recent work.

8 Conclusions

In this paper we have presented a novel approach to reason about assignments in recursive data structures. We have shown how recursive pointer structures can be described in terms of paths obtained by a series of field accesses. We have provided a formal model of these paths as multidot expressions and we have proved a set of rules that describe how an assignment can affect them. Using these rules we have derived separation and interference lemmas for lists and verified an in-place list reversal algorithm. A complete formalisation of the presented work has been carried out in the PVS theorem prover. We have also shown how to apply this approach to reason about other data structures and we have compared it with the standard inductive approach.

Acknowledgments. The authors would like to thank Marko van Eekelen and Sjaak Smetsers for their insightful comments on a draft version of this paper and the anonymous reviewers for their comments.

References

1. Berdine, J., Calcagno, C., O'Hearn, P.: Smallfoot: Modular automatic assertion checking with separation logic. In: de Boer, F., Bonsangue, M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 115–137. Springer, Heidelberg (2006)
2. Blanco, J., Pablo, C.: A semantics for proving class correctness (2005) (unpublished)

3. Bornat, R.: Proving pointer programs in Hoare logic. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 102–126. Springer, Heidelberg (2000)
4. Bundy, A., Basin, D., Hutter, D., Ireland, A.: Rippling: meta-level guidance for mathematical reasoning. Cambridge University Press, Cambridge (2005)
5. Burstall, R.: Some techniques for proving correctness of programs which alter data structures. In: Machine Intelligence, vol. 7, pp. 22–50. Edinburgh University Press (1972)
6. Distefano, D., Filipović, I.: Memory leaks detection in Java by bi-abductive inference. In: Rosenblum, D., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 278–292. Springer, Heidelberg (2010)
7. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
8. Hoare, C.A.R., Jifeng, H.: A trace model for pointers and objects. In: Guerraoui, R. (ed.) ECOOP 1999. LNCS, vol. 1628, pp. 1–17. Springer, Heidelberg (1999)
9. Hubert, T., Marché, C.: Separation analysis for deductive verification. In: Damm, W., Hermanns, H. (eds.) HAV 2007: Heap Analysis and Verification, Braga, Portugal, pp. 81–93 (2007)
10. Marché, C., Paulin-Mohring, C.: Reasoning about Java programs with aliasing and frame conditions. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 179–194. Springer, Heidelberg (2005)
11. Mehta, F., Nipkow, T.: Proving pointer programs in higher-order logic. *Information and Computation* 199, 200–227 (2005)
12. Meyer, B.: Towards practical proofs of class correctness. In: Bert, D., Bowen, J.P., King, S., Waldén, M. (eds.) ZB 2003. LNCS, vol. 2651, pp. 359–387. Springer, Heidelberg (2003)
13. Owre, S., Shankar, N., Rushby, J., Stringer-Calvert, D.: PVS language reference (version 2.4). Technical report, Computer Science Laboratory, SRI International (2001)
14. Reynolds, J.: Separation logic: A logic for shared mutable data structures. In: LICS 2002: Logic in Computer Science, pp. 55–74. IEEE Computer Society, Los Alamitos (2002)
15. Rosenberg, S., Banerjee, A., Naumann, D.A.: Local reasoning and dynamic framing for the composite pattern and its clients (to appear)
16. Tamalet, A.: Yet another semantics for proving class correctness. Master’s thesis, Universidad Nacional de Rosario, Argentina (2006)
17. Tuerk, T.: A formalisation of Smallfoot in HOL. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 469–484. Springer, Heidelberg (2009)
18. von Henke, F.W., Pfab, S., Pfeifer, H., Rueß, H.: Case studies in meta-level theorem proving. In: Grundy, J., Newey, M.C. (eds.) TPHOLs 1998. LNCS, vol. 1479, pp. 461–478. Springer, Heidelberg (1998)

Specification of a Localization Component Driven by a Goal-Based Approach: Some Lessons We Learned

Abderrahman Matoussi, Frédéric Gervais, and Régine Laleau

Université Paris-Est, LACL, IUT Sénart Fontainebleau,
Dpt Informatique, Route Hurtault, 77300 Fontainebleau, France
{abderrahman.matoussi, frederic.gervais, laleau}@u-pec.fr

Abstract. The transition from the requirements phase to the formal specification phase is one of the most painful steps in software development. Up to now, no well-defined process to build initial formal models has been proposed. We have proposed a method in which initial formal models are built incrementally, driven by a goal-based approach. This paper aims at sharing the salient points of our experience to specify a localization component. We discuss the benefit of using a goal-based modeling to obtain an abstract Event-B specification.

Keywords: Requirements engineering, Formal specification, Event-B, KAOS, Localization component.

1 Introduction

Employing formal methods for critical systems specification is steadily growing from year to year. They have shown their ability to produce and improve such systems for large industrial problems such as Paris metro line 14 [5] or Roissy Val [6] using the B method [1]. With most formal methods, an initial mathematical model can be often refined in multiple steps, until the final refinement contains enough details for an implementation. Most of the time, the initial model is built from the description obtained by the requirements analysis. This requires a high level of competence and a lot of practice, especially as there is no well-defined process to assist designers. Therefore, it is difficult to fully comprehend the correspondence between requirements and initial formal specifications. Indeed, the validation of this initial formal specification is very difficult due to the inability for customers to understand formal models. Moreover, it is hard for designers to link them with the initial requirements. Unfortunately, an initial formal model may not be a correct realization of the requirements.

We have defined a method [10] to cope with this problem using a Goal Oriented Requirements Engineering (GORE) approach [3] and the Event-B formal method [2]. The main objective is to propose a constructive approach in which abstract Event-B models are built incrementally from GORE goal models. Applying Requirements Engineering (RE) methods at the very beginning of a design

process and before using formal methods can be interesting since these methods provide a rich way of structuring and documenting the entire requirements documents. Among them, the GORE paradigm is particularly well-suited for requirements engineering since it is nearer to the way humans think and it is easy to understand by all stakeholders. Moreover, it offers some benefits such as (i) providing the rationale for requirements and explaining them to stakeholders; (ii) identifying the responsibilities and the system boundaries.

In RE methods, two kinds of requirements are identified. Functional requirements specify the functions that the system-to-be must be able to perform and non-functional requirements capture properties or constraints under which the system-to-be must operate, such as performance, quality, security concerns. Generally existing RE methods build models where functional and non-functional requirements are closely related. However it appears in practice that these two kinds of requirements do not evolve over time in the same way and that functional requirements are more stable than non-functional ones. This is why our method is based on the definition of three models. The first one describes functional requirements and is the base for deriving formal specifications in Event-B. The second one describes non-functional requirements and the last one describes the impacts of non-functional requirements on functional requirements. In this paper, we focus on the functional requirements model and the derivation of formal specifications. We have chosen the KAOS [4] GORE method to specify this model, mainly because both KAOS and Event-B advocate the use of refinement techniques and have the ability to model both the system and its environment.

This article describes the application of our method in the framework of a research project, called TACOS [11]. We present here an experience with the specification of a localization software component that used GPS, Wifi and sensors technologies. The remainder of this paper is organized as follows. Section 2 overviews the KAOS concepts that we consider in our approach and the Event-B formal method. Sections 3 presents and illustrates our proposed approach through the specification of a localization software component. Relevant issues and related work are discussed in Sections 4 and 5. We conclude our paper in Section 6 with an outline of future work.

2 Background

In this section, we briefly introduce KAOS and Event-B.

2.1 KAOS Method

KAOS (Keep All Objectives Satisfied) [4] is a goal-based requirements engineering method. It provides five complementary sub-models describing the system and its environment: a goal model, an agent model, an operation model, a behavior model and an object model. In this article, we focus on the goal model where the main concept is the concept of goal. A goal is a prescriptive statement of intent the system should satisfy through cooperation of its agents [4].

KAOS identifies two types of goals: behavioral and soft goals. Behavioral goals prescribe intended system behaviors declaratively whereas soft goals prescribe preferences among alternative system behaviors. Behavior goals can be specialized into *Achieve* and *Maintain* goals. An *Achieve* goal specifies a property that the system will achieve some time in the future, whereas a *Maintain* goal specifies a property that must hold at all times in the future. Functional goals can be considered as *Achieve* goals.

A goal model is an AND/OR graph where higher-level goals can be refined into lower-level sub-goals, and then, recursively, into low-level sub-goals that lead to the satisfaction of requirements of the system-to-be. When a goal is AND-refined into sub-goals, all of them must be satisfied for the parent goal to be satisfied. It is possible to precise a specific tactic associated to the AND refinement: the MILESTONE refinement tactic that consists in identifying milestone states that must be sequentially satisfied for the parent goal to be satisfied. When a goal is OR-refined, the satisfaction of one of them is sufficient for the satisfaction of the parent goal. A goal which cannot be further refined is assignable to an agent. An agent is an active system component which plays some role in goal satisfaction. A goal assignable to an agent is called a requisite. A requisite which is placed under the responsibility of an agent of the system to-be is a requirement, whereas a requisite which is placed under the responsibility of an agent in the environment of the system is called an expectation.

2.2 Event-B Method

Event-B [2], an evolution of the classical B method [1], is a formal method for modeling discrete systems by refinement. An Event-B model can be described in terms of two basic constructs:

- **The context:** it provides axiomatic properties of Event-B models. It contains the static part of a model such as carrier sets, constants, axioms and theorems. Carrier sets are similar to types but both, carrier sets and constants, can be instantiated. Axioms describe properties of carrier sets and constants. Theorems are derived properties that can be proved from the axioms. Proof obligations associated with contexts are straightforward: the stated theorems must be proved.
- **The machine:** it contains the dynamic part such as variables, invariants, theorems, events and variants. Variables v define the state of a machine. Possible state changes are described by means of events. Each event is composed of a guard $G(t, v)$ and an action $S(t, v)$, where t are local variables the event may contain. Guards state the necessary condition under which an event may occur, and actions describe how the state variables evolve when the event occurs. The correctness of an Event-B model is defined by an invariant property under which every state in the system must satisfy. Every event in the system must be shown to preserve this invariant. To verify this requirement, *the invariant preservation* proof obligation has been defined.

It is also important to indicate that the most important feature provided by Event-B is its ability to stepwise refine specifications. Refinement is a process that often transforms an abstract and non-deterministic specification into a concrete and deterministic system that preserves the functionality of the original specification. During the refinement, event descriptions are rewritten to take new variables into account. This is performed by strengthening their guards and adding substitutions on the new variables. New events that only assign the new variables may also be introduced. Notice that the state of the abstract machine is related to the state of the concrete machine by a gluing invariant $J(v, w)$, where v are the variables of the abstract machine and w the variables of the concrete machine. In addition to *the invariant preservation*, other proof obligations (POs) are generated to ensure the correctness of the refinement with respect to the abstract machine: (i) *the guard strengthening* ensures that the concrete guard is stronger than the abstract one. In other words, it is not possible to have the concrete version enabled whereas the abstract one would not. The term "stronger" means that the concrete guard implies the abstract guard. (ii) *the correct refinement* ensures that the concrete event transforms the concrete variables in a way which does not contradict the abstract event. Event-B is provided with tool support in the form of an Eclipse-based IDE called Rodin [7].

3 Experience with the Specification of a Localization Software Component

A localization system is a critical part of a land transportation system. Many positioning systems have been proposed over the last years. GPS, one of the most widely used positioning system, is perhaps the best-known. This system belongs to the GNSS (Global Navigation Satellite Systems) family which also regroups GALILEO or GLONASS. Positioning systems are often dedicated to a particular environment; the GNSS technology, for example, generally does not work indoors. To resolve these problems, numerous alternatives relying on very different technologies have arisen. These last years, Wireless LAN such as IEEE 802.11 networks have been considered by numerous location systems. These systems use the radio signal strength to determine the physical location. Localization systems can therefore be designed using various technologies like wireless personal networks such as Wifi or Bluetooth [17][18], GNSS repeaters or visual landmarks.

When elaborating a goal model, the main difficulties are first to identify goals and then to specify links between these goals. Often, requirements engineers need to search through preliminary documents in order to extract goals (key properties) using a number of heuristics (asking HOW and WHY questions...) detailed in [4]. Figure 1 shows the obtained KAOS goal model of a localization component thanks to heuristics. For example, a HOW question about the goal G would then lead to the goals G_1 , G_2 and G_3 . This KAOS goal model contains: (i) high-level goals; (ii) refinement links denoted by a bubble linking the parent goal with an arrow, and the child goals with regular lines; (iii) requirements

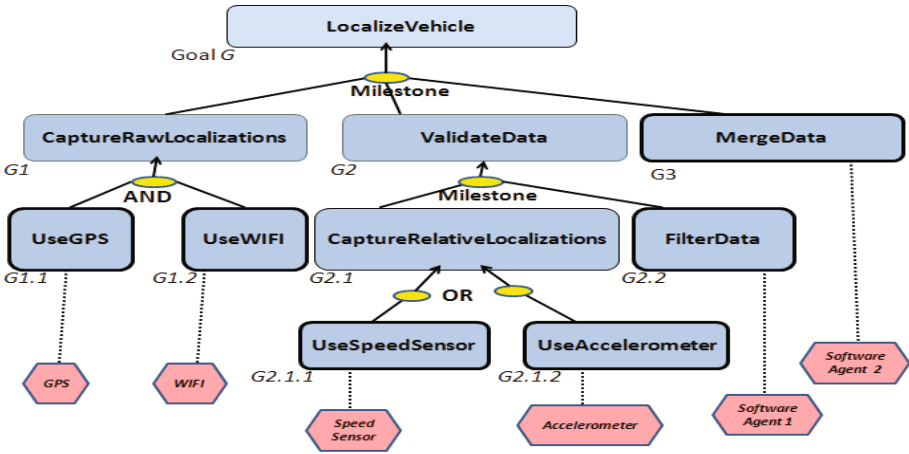


Fig. 1. KAOS goal model of a localization component

and their software responsibility agents; (iv) expectations and their environment responsibility agents (GPS, WIFI, sensor and accelerometer).

Each goal is described informally in natural language. As we deal with functional goals, we consider only *Achieve* goals of KAOS. An *Achieve* goal prescribes intended behaviors where some target condition must sooner or later hold whenever some other condition holds in the current system state. An *Achieve* goal in KAOS is denoted as follows where *CurrentCondition* is optional (said otherwise, it can be true):

$$[if \textit{CurrentCondition} \textit{ then}] \textit{sooner-or-later TargetCondition}.$$

The approach we propose [10] starts from a KAOS goal model describing the functional requirements of a system and derives an abstract Event-B specification. The crux of our transformation is to express each KAOS functional goal as an Event-B event, where: (i) the current condition of this goal is considered as the guard; (ii) the **then** part encapsulates the target condition of this goal. Afterwards, we have transformed into Event-B the different KAOS goal refinement patterns (milestone, AND, OR) used to generate a KAOS goal hierarchy. The main idea is that each level in the hierarchy of the KAOS goal graph is represented as an Event-B model that refines the Event-B model related to the previous level of the hierarchy. Then, we have proposed an Event-B transformation rule for each KAOS refinement pattern associated to functional goals and we have identified for each one the systematic proof obligations. A formal argumentation of the different identified proof obligations is detailed in [10]. In the next sections, we will use these different Event-B refinement relations and additional custom-built proof obligations in order to transform the different refinement levels of the goal model.

3.1 Abstract Model

Let us start by the high-level goal G which is defined as follows:

Goal G : Achieve [LocalizeVehicle]

InformalDef: The vehicle must be localized.

We associate an Event-B model *Localization*, in Figure 2, to this most abstract level of the hierarchy of the KAOS goal graph. In this Event-B model, we will have an event called **LocalizeVehicle** that will translate the goal G ; i.e. it describes the “property” of the goal G , in terms of generalized substitutions. Localizing a vehicle consists in obtaining an *estimated_loc* which is a pair of latitude and longitude. At this level of abstraction, it is not necessary to precise the way this information is calculated. Thus, we use the non-deterministic generalized substitution through the symbol $:\in$ which specifies an unbounded choice. So, *estimated_loc* can take any value in the sets $(LATITUDE \setminus \{null\})$ and $(LONGITUDE \setminus \{null\})$. The *null* value serves just to initialize the system through the **initialization** event¹. Notice that at this abstraction level, the event **LocalizeVehicle** can always occur. Hence, its guard is always *true*. Sets in uppercase are abstract sets used to type the variables. They are described in the Event-B context *TypeSets* (see in Figure 3).

```

MACHINE Localization
SEES TypeSets
VARIABLES
  estimated_loc
INVARIANTS
  inv1 : estimated_loc ∈ LATITUDE × LONGITUDE
EVENTS
Initialisation
  begin
    act1 : estimated_loc := null ↦ null
  end
Event LocalizeVehicle ≐
  begin
    act1 : estimated_loc :∈ (LATITUDE \ {null}) × (LONGITUDE \ {null})
  end
END

```

Fig. 2. The abstract model

3.2 First Refinement

The goal G is refined into three sub-goals according to the milestone goal refinement pattern (identifying milestone states from left to right):

¹ Up to now, the initialization part, variables, invariants and contexts are manually completed by the designer. It would be possible to derive them from domain models in KAOS.

```

CONTEXT TypeSets
SETS
  SUBCOMPONENTS
  SUBSENSORS
CONSTANTS
  gps, wifi, LATITUDE, LONGITUDE
  null, speed, accel
AXIOMS
  axm1 : partition(SUBCOMPONENTS, {gps}, {wifi})
  axm2 : LATITUDE =  $\mathbb{N} \cup \{null\}$ 
  axm3 : LONGITUDE =  $\mathbb{N} \cup \{null\}$ 
  axm4 : partition(SUBSENSORS, {speed}, {accel})
END

```

Fig. 3. The Event-B context *TypeSets*

Goal G_1 : Achieve [CaptureRawLocalizations]

InformalDef: Firstly, several sets of raw localization data are captured by using different technologies.

Goal G_2 : Achieve [ValidateData]

InformalDef: Then, all the sets of raw localization data will be validated and controlled.

Goal G_3 : Achieve [MergeData]

InformalDef: Finally, all the validated data will be merged in order to obtain the final localization.

Similarly, we associate an Event-B refinement model *Localization1*, in Figure 4, to this first level of the hierarchy of the KAOS goal graph. The sub-goals G_1 , G_2 and G_3 are represented by three Event-B events **CaptureRawLocalizations**, **ValidateData** and **MergeData**, respectively. The first one returns a set of couples (latitude, longitude), one for each component used for localizing a vehicle. The second one validates the returned set of couples by choosing the acceptable values. The final one returns the final localization calculated from the returned values of the event **MergeData**.

In Event-B, often the information about such event ordering has to be embedded into guards and event actions with the downside of extra model variables. For that, we have chosen to explicitly reproduce KAOS goal ordering in an Event-B model by proposing a syntactic extension of the Event-B refinement proof rule in order to provide a way to refine an abstract event by a sequence of new events. Hence, the abstract event **LocalizeVehicle** is refined as follows:

(**CaptureRawLocalizations**; **ValidateData**; **MergeData**) Refines
LocalizeVehicle

```

MACHINE Localization1
REFINES Localization
SEES TypeSets
VARIABLES
    subcomponents_loc, validated_loc, merged_loc
INVARIANTS
    inv1: subcomponents_loc ∈ SUBCOMPONENTS → (LATITUDE ×
        LONGITUDE)
    inv2: validated_loc ∈ SUBCOMPONENTS → (LATITUDE × LONGITUDE)
    inv3: merged_loc ∈ LATITUDE × LONGITUDE
    inv4: estimated_loc = merged_location
EVENTS
Initialisation
    begin
        act2: subcomponents_loc :∈ SUBCOMPONENTS → ({null} × {null})
        act3: validated_loc :∈ SUBCOMPONENTS → ({null} × {null})
        act4: merged_loc := null ↦ null
    end
Event CaptureRawLocalizations ≐
    begin
        act1: subcomponents_loc :∈ SUBCOMPONENTS → ((LATITUDE \ {null}) ×
            (LONGITUDE \ {null}))
    end
Event ValidateData ≐
    when
        grd1: subcomponents_loc ∈ SUBCOMPONENTS → ((LATITUDE \ {null}) ×
            (LONGITUDE \ {null}))
    then
        act1: validated_loc :∈ P1(subcomponents_loc)
    end
Event MergeData ≐
    when
        grd1: validated_loc ∈ P1(subcomponents_loc)
        grd2: subcomponents_loc ∈ SUBCOMPONENTS → ((LATITUDE \ {null}) ×
            (LONGITUDE \ {null}))
    then
        act1: merged_loc :∈ (LATITUDE \ {null}) × (LONGITUDE \ {null})
    end

```

Fig. 4. First Event-B refinement model

In addition to *the feasibility proof obligation*², this kind of refinement requires to discharge these different proof obligations:

- *Two ordering constraints* express the “milestone” characteristic between the Event-B events. These two proof obligations are discharged: (i) the action of **CaptureRawLocalizations** implies the guard of **ValidateData** (ii) the action of **ValidateData** implies the guard of **MergeData**.
- *One “guard strengthening” PO* is also discharged since the first event in the sequence (**CaptureRawLocalizations**) has a guard (*true*) that implies the abstract guard (*true*).

² It ensures that each event must also be feasible, in a sense that an appropriate new state v' must exist for some given current state v under the invariant $I(v)$.

- One “correct refinement” PO is also proved since the last event in the sequence (**MergeData**) has a postcondition that implies the abstract postcondition under the gluing invariant $inv4$. The sequence of concrete events transforms the concrete variables in a way which does not contradict the abstract event.

3.3 Second Refinement

Now, we consider the second level of the hierarchy of the KAOS goal graph. In the same way, a refinement Event-B model *Localization2*, in Figure 5, is created and must refine the previous model *Localization1*. This second refinement Event-B model will encapsulate two KAOS refinement patterns:

Second Refinement: Applying the AND Goal Refinement Pattern. The goal $G1$ is AND-refined into two sub-goals; i.e. the conjunction of the sub-goals is sufficient to establish the satisfaction of the parent goal. This refinement specifies the kind of technology used to obtain localization data.

Goal $G_{1.1}$: Achieve [UseGPS]
InformalDef: A GPS system is used.

Goal $G_{1.2}$: Achieve [UseWIFI]
InformalDef: A wireless technique is used.

The sub-goals $G_{1.1}$ and $G_{1.2}$ are represented by two Event-B events **UseGPS** and **UseWIFI** by using the same transcription rules as for the event **LocalizeVehicle** in the abstract model. Since the goal G_1 is refined into two sub-goals $G_{1.1}$ and $G_{1.2}$ according to the AND goal refinement pattern, the execution of the corresponding new events must not necessary follows a specific order. For that, our idea (inspired from Process Algebra [20]) is that these events (**UseGPS** and **UseWIFI**) are executed in an arbitrary order: either **UseGPS;UseWIFI** or **UseWIFI;UseGPS**. This corresponds to the semantics of the interleave operator in process algebra. Of course, we must ensure that this AND refinement manipulate only *disjoint set of variables*. Hence, we have proposed syntactic extension of the Event-B refinement proof rule in order to refine an abstract event by the interleaving of all the new events as follows:

(**UseGPS** ||| **UseWIFI**) Refines **CaptureRawLocalizations**

In addition to *the feasibility proof obligation*, the following proof obligations must be discharged in order to prove such refinement:

- Two “guard strengthening” POs are discharged since the concrete guard of the interleaved events (**UseGPS** ||| **UseWIFI**) implies the abstract guard (*true*) of **CaptureRawLocalizations**. In fact, the concrete guard is always *true* (if we execute **UseGPS** at first or if we execute **UseWIFI** at first).

```

MACHINE Localization2
REFINES Localization1
SEES TypeSets
VARIABLES
  gps_loc, wifi_loc, sensors_loc, kept_loc
INVARIANTS
  inv1 : gps_loc ∈ {gps} → (LATITUDE × LONGITUDE)
  inv2 : wifi_loc ∈ {wifi} → (LATITUDE × LONGITUDE)
  inv3 : subcomponents_loc = gps_loc ∪ wifi_loc
  inv4 : sensors_loc ∈ SUBSENSORS ↔ (LATITUDE × LONGITUDE)
  inv5 : kept_loc ∈ SUBCOMPONENTS ↔ (LATITUDE × LONGITUDE)
  inv6 : validated_loc = kept_loc
EVENTS
Initialisation
  begin
    act6 : gps_loc :∈ {gps} → ({null} × {null})
    act7 : wifi_loc :∈ {wifi} → ({null} × {null})
    act8 : sensors_loc :∈ SUBSENSORS → ({null} × {null})
    act9 : kept_loc :∈ SUBCOMPONENTS → ({null} × {null})
  end
Event UseGPS ≐
  begin
    act1 : gps_loc :∈ {gps} → ((LATITUDE \ {null}) × (LONGITUDE \ {null}))
  end
Event UseWIFI ≐
  begin
    act1 : wifi_loc :∈ {wifi} → ((LATITUDE \ {null}) × (LONGITUDE \ {null}))
  end
Event CaptureRelativeLocalizations ≐
  when
    grd1 : subcomponents_loc ∈ SUBCOMPONENTS → ((LATITUDE \ {null}) ×
      (LONGITUDE \ {null}))
  then
    act1 : sensors_loc : |(sensors_loc' ∈ SUBSENSORS ↔ ((LATITUDE \
      {null}) × (LONGITUDE \ {null}))) ∧ sensors_loc' ≠ ∅
  end
Event FilterData ≐
  when
    grd1 : sensors_loc ∈ SUBSENSORS ↔ ((LATITUDE \ {null}) ×
      (LONGITUDE \ {null})) ∧ sensors_loc ≠ ∅
  then
    act1 : kept_loc :∈ P1(subcomponents_loc)
  end
END

```

Fig. 5. Second Event-B refinement model

- One “correct refinement” PO is also proved since the conjunction of the two concrete postconditions implies the abstract postcondition under the gluing invariant *inv3*. Hence, this ensures that *subcomponents_loc* is a total function (see the postcondition of the abstract event *CaptureRawLocalizations*).

Second Refinement: Applying the Milestone Goal Refinement Pattern

On the other hand, the goal G_2 is refined into two sub-goals according to the milestone goal refinement pattern (the order is from left to right):

Goal $G_{2.1}$: Achieve [CaptureRelativeLocalizations]

InformalDef: At first, several sets of relative localization data are captured by using different technologies.

Goal $G_{2.2}$: Achieve [FilterData]

InformalDef: Then, all the sets of raw localization data will be filtered.

All these subgoals are translated into new events using the same rules as for **LocalizeVehicle** in the abstract Event-B model. As for the first refinement, the abstract event **ValidateData** is refined by the *sequence* of all the new events (**CaptureRelativeLocalizations**, **FilterData**) as follows:

(**CaptureRelativeLocalizations** ; **FilterData**) Refines **ValidateData**

We have also discharged the different proof obligations related to the milestone refinement such as *the ordering constraint*, *the “guard strengthening”* and *the “correct refinement”*.

3.4 Third Refinement

The goal $G_{2.1}$ is OR-refined in two subgoals:

Goal $G_{2.1.1}$: Achieve [UseSpeedSensor]

InformalDef: The Vehicle may use a speed sensor system.

Goal $G_{2.1.2}$: Achieve [UseAccelerometer]

InformalDef: Or, it may use the accelerometer system.

Similarly, a refinement Event-B model *Localization3*, in Figure 6, is associated to this third level of the hierarchy of the goal graph. All these subgoals are transformed into new Event-B events (**UseSpeedSensor**, **UseAccelerometer**) by using the same transcription rules as for the event **LocalizeVehicle** in the abstract Event-B model. Since the satisfaction of exactly one KAOS sub-goal implies the satisfaction of the parent goal, we propose to refine the abstract event **CaptureRelativeLocalizations** as follows:

(**UseSpeedSensor** XOR **UseAccelerometer**) Refines
CaptureRelativeLocalizations

This Event-B refinement semantics is quite close to the same one proposed by Rodin 7 if we consider that each event refines the abstract event. Hence, the

```

MACHINE Localization3
REFINES Localization2
SEES TypeSets
VARIABLES
  sensors_loc, speed_loc, accel_loc
INVARIANTS
  inv1 : speed_loc ∈ {speed} → (LATITUDE × LONGITUDE)
  inv2 : accel_loc ∈ {accel} → (LATITUDE × LONGITUDE)
  inv3 : (sensors_loc = speed_loc) ∨ (sensors_loc = accel_loc) ∨ (sensors_loc =
    speedloc ∪ accel_loc)
EVENTS
Initialisation
  begin
    act10 : speed_loc := {speed} → ({null} × {null})
    act11 : accel_loc := {accel} → ({null} × {null})
  end
Event UseSpeedSensor ≐
refines CaptureRelativeLocalizations
  when
    grd1 : subcomponents_loc ∈ SUBCOMPONENTS → ((LATITUDE \ {null}) ×
      (LONGITUDE \ {null}))
    grd2 : accel_loc ∈ {accel} → ({null} × {null})
  then
    act1 : speed_loc, sensors_loc : | (speed_loc' ∈ {speed} → ((LATITUDE \
      {null}) × (LONGITUDE \ {null}))) ∧ sensors_loc' = speed_loc'
  end
Event UseAccelerometer ≐
refines CaptureRelativeLocalizations
  when
    grd1 : subcomponents_loc ∈ SUBCOMPONENTS → ((LATITUDE \ {null}) ×
      (LONGITUDE \ {null}))
    grd2 : speed_loc ∈ {speed} → ({null} × {null})
  then
    act1 : accel_loc, sensors_loc : |(accel_loc' ∈ {accel} → ((LATITUDE \ {null}) ×
      (LONGITUDE \ {null}))) ∧ sensors_loc' = accel_loc'
  end
END

```

Fig. 6. Third Event-B refinement model

proof obligations (“guard strengthening” and “correct refinement”) could be discharged by the current version of the Rodin automatic theorem prover [7]. This is due essentially to the gluing invariant *inv3* and the individual guards of each event. However, we require to express two additional proof obligations ensuring that only one event (either **UseSpeedSensor** or **UseAccelerometer**) but not both can be executed. These two proof obligations are discharged since (i) the postcondition of **UseSpeedSensor** forbids the guard of **UseAccelerometer** to be triggered; (ii) the postcondition of **UseAccelerometer** forbids the guard of **UseSpeedSensor** to be triggered.

This is the last step of refinement since all the goals are either *requirements* or *expectations* (see Figure 11).

3.5 Synthesis

Regarding the proof activity linked to the use of Event-B, in addition to the classical proof obligations such as feasibility proof obligations, the resulting Event-B specification led to a number of 16 additional custom-built proof obligations (relatively simple) that have been proved totally. Most of these proof obligations can be automatically discharged by the Rodin tool.

Unfortunately, discharging the different proof obligations is not sufficient in order to validate the conformance of the specification to original requirements. In fact, we can never ensure that the expression of the Event-B event corresponds exactly to the expression of the related goal since this latter is informal. For that, we can use an animation technique to validate the derived formal specification against original customers' requirements. This animation step not only indicates deviations from original requirements right on the spot but also helps fixing some specification errors. The reader can refer to [16] for more details. For that, ProB [14] may be a very useful validation tool since its automated animation facilities allow users to animate their specifications; i.e. gain confidence in their specifications.

The obtained abstract Event-B model is then further refined towards an implementation. It concerns only the Event-B events corresponding to requirements assigned to software agents. Otherwise, an expectation (assigned to an external agent) is a property required on the environment and its corresponding Event-B event will not be implemented in the software-to-be. More precisely, in our case study, the Event-B events **UseGPS**, **UseWIFI**, **UseSpeedSensor** and **UseAccelerometer** are not refined since they correspond to goals of type *expectation*. They are implemented by hardware components (GPS, WIFI components...) in a vehicle. Only the Event-B events **FilterData** and **MergeData** are further refined. For example, the refinement of the Event-B event **MergeData** leads to a software that implements the algorithm chosen to realize the fusion (thanks to the B operation **OPMerge**) as shown in Figure 7. At first, this operation recovers all the raw localization data from both GPS and WIFI thanks to the call of operations *get_lat* and *get_long*. Then, the call of the operation *get_pond* serves to verify if the returned values of GPS and WIFI are validated or no. If these values are validated, then *get_pond* returns a weighting value set to 1 (0 otherwise). Finally, the operation calculates the final latitude and longitude based on the different weighting values.

An interesting result is that the link between the B operation **OPMerge** and the abstract Event-B event **MergeData** (see Figure 4) can be ensured. While the abstract event **MergeData** describes the properties that the final program must fulfill, the B operation **OPMerge** describes the algorithm contained in the program. Hence, **MergeData** describes the way by which we can eventually judge that the final program **OPMerge** is correct: (i) the call of the operations *get_lat* and *get_long* ensures the second guard of **MergeData**; (ii) the call of the operation *get_pond* ensures the first guard of **MergeData**; (iii) the final result of the operation **OPMerge** (*lat*, *long*) satisfies the post-condition of **MergeData**.

```

lat, long ← OPMerge =
VAR lat_gps, long_gps, lat_wifi, long_wifi, ponderation_gps, ponderation_wifi
IN
  lat_gps := get_lat(gps_loc) ||
  long_gps := get_long(gps_loc) ||
  ponderation_gps := get_pond(gps_loc) ||
  lat_wifi := get_lat(wifi_loc) ||
  long_wifi := get_long(wifi_loc) ||
  ponderation_wifi := get_pond(wifi_loc) ;

  lat := ((lat_gps * ponderation_gps) + (lat_wifi * ponderation_wifi))
         / (ponderation_gps + ponderation_wifi) ||
  long := ((long_gps * ponderation_gps) + (long_wifi * ponderation_wifi))
         / (ponderation_gps + ponderation_wifi)
END

```

Fig. 7. The B operation **OPMerge**

4 Lessons Learned

The first conclusion that can be drawn from this experience concerns the validity of our approach. The elaboration of the specification of the localization component has been carried out with specialists of the domain, with no skills in formal and GORE methods. Four observations can be stated. Firstly, the visual dimension (boxes, arrows...) can assist people to better understand requirements. Secondly, GORE methods help designers to structure the Event-B specification and to choose the relevant refinement level to introduce the different Event-B events. Thirdly, these methods aid designers to correctly build guards of each Event-B event. For instance, at the most abstract level, the guard of **LocalizeVehicle** is set to *TRUE* to express that the event is always feasible. The definitive guard is built during the refinement process. Finally, the employment of GORE encourages designers to consider both the system and its environment.

In our view, explicitly expressing the notion of ordering, interleaving and exclusivity between events is a useful addition to the Event-B refinement semantics. Up to now, encoding these notions is possible in the current state of Event-B using extra model variables (flags) embedded into guards and event actions. The drawback of such extra model variables is the entanglement of order encoding and functional specification. This entanglement makes traceability between requirements and specification complicated. The proposed Event-B refinement extension (without extra model variables) makes such Event-B models more readable and may allow a proof obligation economy. This last benefit is under study and must be normally confirmed by further case studies.

5 Related Work

Most of the existing work aiming at establishing links between requirements models and formal methods consider KAOS and B or VDM++. The work of [13]

associates a B machine to each KAOS agent since agents are the active entities able to perform operations. For that, all the KAOS operations that an agent has to perform are represented by B operations. Moreover, all *maintain* goals under the agent responsibility are translated as invariants of the corresponding B machine. The authors of [9] provides means for transforming the security requirements model built with KAOS to an equivalent one in B. This abstract B model is then refined using non-trivial B refinements that generate design specifications conforming to the initial set of security requirements. Recently, [15] presents a constructive verification-based approach for operationalizing requirements into specifications expressed in an extended Event-B formalism. We can also point out a work [12] proposing an automatically generator that transforms an extend KAOS model into VDM++ specifications. The generator connects operations in KAOS to those in VDM++, and entities in KAOS to objects or types in VDM++. The generated specification contains implicit operations consisting of pre- and post-conditions, inputs, and outputs of operations.

Nevertheless, the reconciliation presented by all of these works remains partial because they don't consider all the parts of the KAOS goal model but only the requirements (operational goals). Consequently, the formal model does not include any information about the non-operational goals and, more important, the type of goal refinement. Yet, non-operational goals play an important role for requirements completeness and pertinence and provide for example the rationale for the requirements that operationalize them. In this paper, we have explored how to cope with this problem using an approach that transform the whole KAOS goal model to abstract Event-B models. Our approach can be considered as complementary to existing ones. Furthermore, what we present can be very useful in practice to systematically verify that all KAOS requirements are represented in the Event-B model.

6 Conclusion and Further Work

This paper presents an application of our method that consists in building a goal-oriented requirements specification to derive an Event-B specification. It confirms that GORE methods provide a possible way of building and structuring formal specifications. A number of future research steps are ongoing. We currently develop the model to represent non-functional goals and their impacts on functional goals [21]. It is inspired from the *i** method [19] and from [23]. The complete method will be defined as an extension of SysML [22]. We have already developed a support tool [21] on the TOPCASED [8] open platform based on Eclipse and started the development of a plug-in between this tool and the RODIN [7] open platform.

Acknowledgment

The work in this paper is partially supported by the TACOS project [11] ANR-06-SETI-017 founded by the french ANR (National Research Agency).

References

1. Abrial, J.R.: *The B-Book: Assigning programs to meanings*. CUP (1996)
2. Abrial, J.R.: *Modeling in Event-B: System and Software Engineering*. CUP (2010)
3. Nuseibeh, B., Easterbrook, S.: Requirements engineering: a roadmap. In: 22nd ACM International Conference on Software Engineering, Future of Software Engineering Track, Limerick, Ireland, pp. 35–46 (2000)
4. van Lamsweerde, A.: *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, Chichester (2009)
5. Behm, P., Benoit, P., Faivre, A., Meynadier, J.-M.: METEOR: A successful application of B in a large project. In: Woodcock, J.C.P., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 369–387. Springer, Heidelberg (1999)
6. Badeau, F., Amelot, A.: Using B as a high level programming language in an industrial project: Roissy val. In: Treharne, H., King, S., Henson, M.C., Schneider, S. (eds.) Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 334–354. Springer, Heidelberg (2005)
7. RODIN - Rigorous Open Development Environment for Complex Systems, <http://rodin.cs.ncl.ac.uk/>
8. TOPCASED, <http://www.topcased.org/>
9. Hassan, R., Bohner, S., El-Kassas, S., Eltoweissy, M.: Goal-Oriented, B-Based Formal Derivation of Security Design Specifications from Security Requirements. In: ARES 2008, Spain, pp. 1443–1450. IEEE Computer Society Press, Los Alamitos (2008)
10. Matoussi, A., Gervais, F., Laleau, R.: An Event-B formalization of KAOS goal refinement patterns. Technical Report TR-LACL-2010-1, LACL, University of Paris-Est (2010), <http://lacl.univ-paris12.fr/Rapports/TR/TR-LACL-2010-1.pdf>
11. TACOS Project. ANR-06-SETIN-017, <http://tacos.loria.fr>
12. Nakagawa, H., Taguchi, K., Honiden, S.: Formal Specification Generator for KAOS. In: ASE 2007, Atlanta, USA, pp. 531–532. ACM, New York (2007)
13. Ponsard, C., Dieul, E.: From Requirements Models to Formal Specifications in B. In: REMO2V 2006, Luxembourg (June 2006)
14. Leuschel, M., Butler, M.J.: ProB: A Model Checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
15. Aziz, B., Arenas, A., Bicarregui, J., Ponsard, C., Massonet, P.: From Goal-Oriented Requirements to Event-B Specifications. In: First Nasa Formal Method Symposium (NFM 2009), Moffett Field, California, USA (April 2009)
16. Mashkoor, A., Matoussi, A.: Towards Validation of Requirements Models. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, p. 404. Springer, Heidelberg (2010)
17. Hallberg, J., Nilsson, M., Synnes, K.: Positioning with bluetooth. In: 10th Int. Conference on Telecommunications (ICT 2003), pp. 954–958 (2003)
18. Royo, J.A., Mena, E., Gallego, L.C.: Locating Users to Develop Location-Based Services in Wireless Local Area Networks. In: UCAM I 2005, Granada, Spain, pp. 471–478 (2005)
19. Yu, E.: Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering. In: RE 1997, pp. 226–235. IEEE Computer Society, Los Alamitos (1997)

20. Sangiorgi, D.: Locality and interleaving semantics in calculi for mobile processes. *Theor. Comput. Sci.* 155, 39–83 (1996)
21. Gnaho, C., Semmak, F.: Une extension SysML pour l'ingénierie des exigences dirigée par les buts. In: *INFORSID 2010*, Marseille, France, pp. 277–292 (May 2010)
22. Laleau, R., Semmak, F., Matoussi, A., Petit, D., Hammad, A., Tatibouet, B.: A first attempt to combine SysML requirements diagrams and B. *Innovations in Systems and Software Engineering* 1-2, 47–54 (2010)
23. Chung, L.: *Non-Functional Requirements In Software Engineering*. Kluwer Academic Publishers, Dordrecht (1999)

A Formal Framework for Specifying and Analyzing Logs as Electronic Evidence

Eduardo Mazza¹, Marie-Laure Potet¹, and Daniel Le Métayer²

¹ Verimag, Centre Équation, 2 avenue de Vignate, F-38610 Gières
{Eduardo.Mazza,Marie-Laure.Potet}@imag.fr

² LICIT, INRIA Grenoble Rhône-Alpes
Daniel.Le-Metayer@inrialpes.fr

Abstract. The issues of logging for determining liability requires to define, prior to a dispute, the logging system and the log analysis in a manner that would determine the parties liable for a predetermined misbehavior of the system. We propose a formal framework for specifying and reasoning about decentralized logs to be used in legal disputes. In addition, we study how previous results can be used in the incremental analysis of larger inputs to obtain precise or approximated results. We illustrate our approach with an example of a travel arrangement service.

1 Introduction

Due to the growing impact of the Information and Communication Technologies on everyday life, and the increasing complexity of computer systems, the logging of these systems raises greater challenges. Logging is a necessity for debugging a system at development time, or after a fault, for identifying security attacks, guaranteeing safety, and establishing liability for software providers. With respect to the last one, Fred B. Schneider pointed [23] that liability determination needs a mature discipline of forensics for computing systems and components. It requires to change software development practices, because, in addition to delivering systems, producers will also need to deliver instruments to show that they behave well.

The use of log as electronic evidence for establishing contractual liability is a challenging problem [18, 6, 12]. Actual solutions that propose formal models to specify liability [17, 10] are more focused on the system models rather than using logs as electronic evidences in cases of litigation. Meanwhile, other works [4, 11, 21, 3] present a well-defined model for analysing properties in logs, but a small effort has been made to specify the liability associated with the log content. Existing research stops short of discussing how one might determine whether the information found in a given log is precise enough to be used for legal disputes.

The LISE project [1] aims to address liability issues from the legal and technical points of view. The goal is to cover the whole chain of “liability engineering”, from

¹ Liability Issues in Software Engineering: <http://licit.inrialpes.fr/lise/>

liability specification (at contract negotiation time) to liability determination (at litigation time). With the purpose of reduce legal uncertainty, the LISE approach incorporates in B2B contracts (between services providers) an agreement about the electronic evidence to be produced and used in case of failures. According to this agreement, the parties commit to build these pieces of evidence, and to rely on them for determining their share of responsibility. We assume that certain conditions (such as, proof of authenticity and integrity) are satisfied by complementary means.

In contrast with others frameworks that focus either in liability [17, 10] or properties verification for logs [4, 11], we propose here an integrated framework that allow us to specify liability, claims, and logs as electronic evidence (Figure 1a). As mentioned before we restrict ourselves in the context of liability defined for a contractual environment. Furthermore, we exploit the central notion of agents (seen as parties implied in a contractual engagement) that organize claims, properties and distributed logs in a very tractable way. Then, we are able to propose a general analysis procedure (Figure 1b) allowing us to evaluate the admissibility of a given claim instance. This procedure is focused on the notion of properties attached to a given claim, rather than general properties describing the behavior of the system [4, 11, 21, 3]. The fact that, in our context, claims are defined a priori allow us to provide a simpler specification for such procedure. We also study the incremental aspects of our procedure and propose alternative solutions to obtain new results based on the results of a previous analysis.

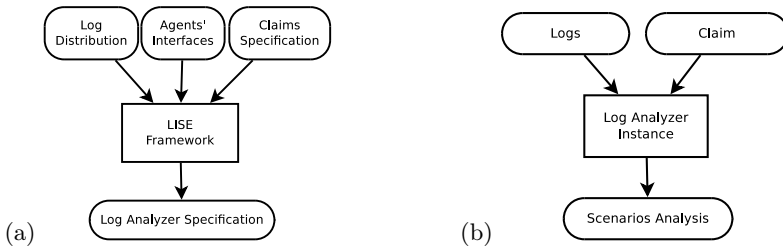


Fig. 1. Framework for liability specification and analysis

This paper is dedicated to the presentation of our framework for the formal representation and analysis of logs. We explore the aspects of the LISE methodology introduced in [15] considering the distribution and analysis of logs. Another previous work [16] studies how different distribution of logs can be classified with relation to their content w.r.t. the level of interest that logging agents have in changing the events in their logs. These can be viewed as complementary results of the present work. Section 2 introduces a motivating example and some useful notations. Section 3 presents our general model to specify logs, log distributions and claims. Section 4 introduces some classical operations on distributed logs, as extraction and merge. Finally, Section 5 gives the specification of our log analysis and its use for claim admissibility together with the studies of incremental results.

2 Case Study and Notations

In this section we present our case study and give a brief background of the B notation used in this paper.

2.1 Case Study

Throughout this paper we will use a travel booking case study as a running example of the different aspects of our framework.

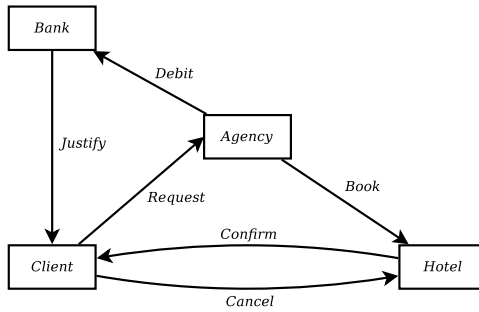


Fig. 2. Travel booking system

In this case study (Figure 2), the client (*Client*) submits a request (*Request*) for a hotel reservation to the travel agency (*Agency*). On the request arrival, *Agency* performs some research to find a hotel that supplies the specifications requested by *Client* (for example, price or hotel location). Having found a hotel, *Agency* sends a request (*Book*) to the hotel (*Hotel*) with the specification of dates and room. *Hotel* confirms the reservation sending a message (*Confirm*) to *Client*. *Client* can cancel the reservation before the reservation date free of charge. To cancel the reservation, *Client* shall send a message (*Cancel*) to *Hotel* informing that s/he no longer desires the room. If *Client* did not cancel the reservation within the time specified, *Agency* makes a debit in the client's account sending a message (*Debit*) to *Bank*. Finally, *Bank* sends a message (*Justify*) to *Client* with the justification for the payment made with his/her account.

From the *Client's* point of view a number of things could go wrong. We consider here two claims:

Example 1. (claim *NoRoom*) Let us consider the case that *Client* submits a request and receives a justification from *Bank*, but when s/he arrives at the hotel there is no information about the reservation in *Hotel's* registry. *Client* complains to *Agency* that s/he received a message from *Bank* and still there is no room available.

Example 2. (claim *LateCancel*) Let us consider another claim where having canceled the reservation, *Client* is still charged by *Agency*. Then, *Client* complains to *Agency* that s/he sent the message to cancel the reservation. On the other side, *Agency* complains that *Client* did not send the message before the reservation date. In this example, the time that the client sends the message can be confused with the time that the message is received by *Hotel*. It is necessary to make clear in the agreement between *Agency* and *Client* how such details will be taken into account.

In such situations, liability can be specified in terms of well-defined properties written in function of events recorded in the logs. Consider, for example, the scenario in Figure 3. In this scenario *Agency* does not send the request of reservation to *Hotel*, but still charges the client by sending a message to *Bank*. *Client* receives the justification from *Bank* and could think that, although s/he has no confirmation from *Hotel* (maybe *Client* does not even know that should receive a confirmation from *Hotel*), the fact that s/he was charged by *Agency* assures the confirmation of the reservation.

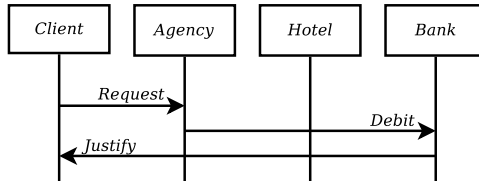


Fig. 3. Sequence diagram for possible scenario where claim *NoRoom* will happen

For the agreement to work it is necessary that the values found in the logs can be trusted by all entities of the system and that it contains no error. In this paper, we assume that this hypothesis is valid and the values that compose the logs correspond exactly to what happened (no duplication, no replication, no loss).

2.2 Notation B

This formal framework is based on the B-method [1]. This methodology was chosen because it allows specifications that are focused both on data and behavior. Another important aspect for our lawyer partners is the industrial status of this approach [5, 2] that is a positive argument in term of trust, a central notion in law.

We give here a short summary of the B notations we use in this paper. Given two sets, A and B , the set of relations between A and B , the set of functions from A to B , and the set of partial functions from A to B are respectively denoted by $A \leftrightarrow B$, $A \rightarrow B$, and $A \rightsquigarrow B$. We note $\mathcal{F}(A)$ all finite subsets of A , $R[U]$ the relational image of U under the relation R (set of all elements that are under relation R with an element of U), and $union(A)$ the generalized union of A (union of all elements of the set A). We denote by f^{-1} the inverse function for f and by $iseq(A)$ the set of

injective sequences of elements of A . A sequence is delimited by the square brackets ('[' and ']') and its elements separated by commas (','). The notation $\lambda x.(P \mid E)$ denotes the function that maps every x that verifies the predicate P into the value of expression E . Finally, for readability, we sometimes omit the declaration of the type of variables whenever we judge the type is easy to infer.

3 Logs and Claims

In this section we present how, in our framework, we specify the expected information about the system, the logs and the claims. The proposed model is based on exchanged messages that are used to represent the interactions between the entities of the system. We will assume here that each message is unique. We will also assume that we are able to identify the origin of exchanged messages. However, our framework can be easily adapted for working without assuming such hypothesis. This model of communication is well adapted to the domain of B2B applications that we mentioned in the introduction, including electronic service and resource provision.

We consider a system consisting of agents participating in some form of interactions described as events, and each event is performed by one of the agents. Particular agents may have the ability to monitor the events performed by itself and other agents and record these events in a log². The agents should agree that the produced logs are liable for representing the event's history.

3.1 System Information

The machine *SystemInfo* specifies a set of agents (*AGENT*) and a set of actions (*ACTION*). Each action is associated with the agent that can execute it by the function *Interface*. The information associated with the system in study is marked in bold.

Example 3. (agents and interface) Let us write the machine *SystemInfo* for our case study such that:

```

MACHINE SystemInfo
SETS
  AGENT = {Client, Agency, Hotel, Bank};
  ACTION = {Request, Book, Debit, Confirm, Justify, Cancel}
CONSTANTS Interface
PROPERTIES
  Interface : ACTION → AGENT ∧
  Interface = {(Request, Client), (Book, Agency), (Debit, Agency),
              (Confirm, Hotel), (Justify, Bank), (Cancel, Client)}
END

```

² Logs can also be produced by some mechanism that does not directly belong to the system, such as protocol sniffers.

The actions specified for a given system may depend on the claims that need to be evaluated. Other actions, such as confirmation of the reservation between *Hotel* and *Agency*, were not considered here to keep the simplicity of the example.

3.2 Logs and Distribution

An event (*EVENT*) is described as a tuple consisting of the type of event (*OP*), the source and destination agents, and the action to be executed. A log file (*LOG_FILE*) is defined as a pair consisting of the set of agents that have their events logged and the sequence of events, recorded in the order of execution.

A log architecture (*LOG_ARCH*) represents a set of logs produced during the execution of the system. The constant *Dist* describes how we have chosen to regroup and distribute logs. Each element $X \in Dist$ represents that a single log file records the events performed by the agents in X . Notice that in this definition the events of an agent might be recorded in more than one log file.

Logs are modeled using the machine *LogModel*.

```

MACHINE LogModel
INCLUDES SystemInfo
SETS OP = {Send, Rec}
CONSTANTS EVENT, LOG_FILE, LOG_ARCH, Dist
PROPERTIES
  EVENT = OP × AGENT × AGENT × ACTION ∧
  /* Log files */
  LOG_FILE = F(AGENT) × iseq(EVENT) ∧
  /* Definition of the chosen log distribution */
  Dist ⊆ F(AGENT) ∧ Dist = {⋯} ∧
  /* Log architectures defined as a set of log files */
  LOG_ARCH = {logs | logs ∈ F(LOG_FILE) ∧ ∀ log.(log ∈ logs ⇒
  agents(log) ∈ Dist)}
END
    
```

In the rest of the paper we use the functions *agents* and *content* that maps every log file into its agents and content respectively. The function *events* maps every log file into the set of events (rather than the sequence) of the log's content. We abuse our notation and assume that *agents* and *events* can also be applied to log architectures: applying *agents* (or *events*) to a log architecture *log_arch* is equivalent to applying *agents* (or *events*) to each log file that belongs to *log_arch* and make the union of the results.

Example 4. (log distribution) Let us consider the following two examples of distributions:

1. $Dist = \{\{\mathbf{Client}\}, \{\mathbf{Agency}\}, \{\mathbf{Hotel}\}, \{\mathbf{Bank}\}\}$
2. $Dist = \{\{\mathbf{Client}, \mathbf{Agency}\}, \{\mathbf{Hotel}\}, \{\mathbf{Bank}\}\}$

In the first example we describe a distribution where each agent is logged independently. In the second example we assume that there is a single log file that records the events performed by *Client* and *Agency*.

3.3 Properties and Claims

In our case study, the agents make an agreement to trust the content of logs, in order to establish liability for a given set of claims. Then it is necessary to express these claims in terms of log events in an unambiguous way. We assume that liability for claims take the form: “if *Prop* holds, then agent defendant is responsible”. This structure is explained in more details in [15].

The machine *Claims* below introduces some general definitions and allows us to declare particular instances of properties and claims.

```

MACHINE Claims
INCLUDES LogModel
CONSTANTS PROP, CLAIM
PROPERTIES
  /* Log Properties */
  PROP = {prop | prop ∈  $\mathcal{F}(AGENT) \times (LOG\_FILE \rightarrow BOOL) \wedge$ 
     $\forall(ags, log).(prop = (ags, log) \Rightarrow ags = agents(log))$ }  $\wedge$ 
    propNoRoom ∈ PROP  $\wedge$  propLateCancel ∈ PROP  $\wedge$  ...  $\wedge$ 
  /* Claims */
  CLAIM = {claim | claim ∈ (AGENT  $\times$  AGENT  $\times$  PROP)  $\wedge$ 
     $\forall(plain, def, prop).(claim = (plain, def, prop) \Rightarrow$ 
       $\{plain, def\} \subseteq agents(prop))$ }  $\wedge$ 
  NoRoom ∈ CLAIM  $\wedge$  NoRoom = (Client, Agency, propNoRoom)  $\wedge$ 
  LateCancel ∈ CLAIM  $\wedge$  LateCancel = (Client, Agency, propLateCancel)
END

```

Due to the distributed nature of logs, the first element of a property explicitly states the agents that are concerned with this property. The second element of the property is a partial function that maps a log file to true or false depending if the log file verifies the property. We use the notation $agents(prop)$ and $val(prop)$ indicating respectively the first and second parts of a property *prop*. In a property it is imposed that the domain of logs matches exactly with the agents concerned by the property. Claims are tuples consisting of a plaintiff, a defendant, and a property that describes the claim, with the plaintiff and the defendant belonging to the agents of the property.

Example 5. (property $prop_{NoRoom}$) Now, consider the claim *NoRoom* described in Section 2.1. Let $prop_{NoRoom}$ be the property that describes the claim where *Client* complains of *Agency* being responsible for charging him/her without booking the reservation:

$$\begin{aligned}
 agents(prop_{NoRoom}) &= \{Client, Agency\} \wedge \\
 val(prop_{NoRoom}) &= \lambda log.(agents(log) = \{Client, Agency\} \mid \\
 & (Send, Client, Agency, Request) \in events(log) \wedge \\
 & (Send, Agency, Bank, Debit) \in events(log) \wedge \\
 & (Send, Agency, Hotel, Book) \notin events(log) \wedge \\
 & pos((Send, Client, Agency, Request), log) < \\
 & pos((Send, Agency, Bank, Debit), log))
 \end{aligned}$$

This definition starts limiting the agents of the property to *Client* and *Agency*. The second part of the definition gives the conditions that verify the property. We use the function $pos(ev, log)$ that maps every event ev and log log such that $ev \in events(log)$, into the position of ev within the sequence $content(log)$.

4 Log Functions

Like others frameworks dedicated to distributed systems we provide some functions for manipulating distributed logs. The functions presented here are based in the well known relation “happened-before” introduced in the early work of Lamport [14].

4.1 Log Extraction

The function *extract* allow us to obtain information contained in a log file concerning a certain group of agents.

Definition 1. (function *extract*) *The partial function $extract : (\mathcal{F}(AGENT) \times LOG_FILE) \mapsto LOG_FILE$ maps every pair (ags, log) such that $ags \subseteq agents(log)$, into log_{ext} with log_{ext} having the following properties:*

1. $agents(log_{ext}) = agents(log)$
2. $events(log_{ext}) = \{ev \mid ev \in events(log) \wedge \exists (ag_1, ag_2, ac).(ag_1 \in agents(log) \wedge ev = (Send, ag_1, ag_2, ac) \vee ev = (Rec, ag_2, ag_1, ac))\}$
3. $\forall (ev_A, ev_B).(ev_A \in events(log_{ext}) \wedge ev_B \in events(log_{ext}) \wedge pos(ev_A, log_{ext}) < pos(ev_B, log_{ext}) \Rightarrow pos(ev_A, log) < pos(ev_B, log))$

The last two properties of Definition 1 respectively state that the extracted log (2.) contains all events that represent messages sent or received by the agent in ags and (3.) respects the order of events in log . In the rest of this paper we use $extract_{ags}(log)$ to denote $extract(ags, log)$.

Example 6. (application of *extract*) Let us imagine the variable $log \in LOG_FILE$ representing the log of *Client* and *Agency* for the scenario described in Figure 3:

$$log = (\{Client, Agency\}, [(Send, Client, Agency, Request), (Rec, Client, Agency, Request), (Send, Agency, Bank, Debit), (Rec, Bank, Client, Justify)])$$

Then, we can use *extract* to obtain the events performed only by *Agency*:

$$extract_{\{Agency\}}(log) = (\{Agency\}, [(Rec, Client, Agency, Request), (Send, Agency, Bank, Debit)])$$

4.2 Log Merge

The relation *merge* produces, for a given log architecture, the set of logs respecting the order of each log file in the architecture and the order of corresponding *Send/Rec* events. Each log produced by this relation represents a scenario describing the sequence of events in the order they were performed.

Definition 2. (relation *merge*) *The relation $merge : LOG_ARCH \leftrightarrow LOG_FILE$ maps every log architecture into logs that will represent all possible total orders for this architecture. That is, for any log_arch and log , such that $(log_arch, log) \in merge$, then:*

1. $agents(log) = agents(log_arch)$
2. $events(log) = events(log_arch)$
3. $\forall log_{ag} \in log_arch \Rightarrow extract_{agents(log_{ag})}(log) = log_{ag}$
4. $\forall (ev_B, ag_1, ag_2, ac). (ev_B \in events(log_arch) \wedge ev_B = (Rec, ag_1, ag_2, ac) \wedge ag_1 \in agents(log_arch) \Rightarrow \exists (ev_A). (ev_A \in events(log_arch) \wedge ev_A = (Send, ag_1, ag_2, ac) \wedge pos(ev_A, log) < pos(ev_B, log)))$

The last two properties in Definition 2 respectively state that the scenario represented by *log* (3.) respects the local order of each log file in *log_arch* and (4.) for every event of type *Rec* if there is a corresponding event of type *Send* then the event of type *Send* precedes the event of type *Rec* in this scenario. That is, the relation *merge* collects all interleaving that respect local and causal orderings.

Example 7. (application of *merge*) Let us, from now on, denote events omitting the sender and receiver and using the simplified notation $(Send, ac)$ to represent an event of the form $(Send, ag_1, ag_2, ac)$ and similar for (Rec, ac) . Let us imagine the distribution *log_arch* composed by the logs of *Client* and *Agency* where:

$$\begin{aligned} log_{Client} &= (\{Client\}, [(Send, Request), (Rec, Justify)]) \\ log_{Agency} &= (\{Agency\}, [(Rec, Request), (Send, Debit)]) \end{aligned}$$

Then, $merge[log_arch]$ ³ produces the set $\{log_1, log_2, log_3\}$ such that:

$$\begin{aligned} log_1 &= \{Client, Agency\}, [(Send, Request), \\ & (Rec, Justify), (Rec, Request), (Send, Debit)] \\ log_2 &= \{Client, Agency\}, [(Send, Request), \\ & (Rec, Request), (Rec, Justify), (Send, Debit)] \\ log_3 &= \{Client, Agency\}, [(Send, Request), \\ & (Rec, Request), (Send, Debit)], (Rec, Justify) \end{aligned}$$

That is, the event $(Rec, Justify)$ can be permuted with the events $(Rec, Request)$ and $(Send, Debit)$ because there is no order constraint between these events. However, we know that all these events shall come after the event $(Send, Request)$ and that the event $(Send, Debit)$ comes after the event $(Rec, Request)$.

³ In this paper we use the notation $merge[log_arch]$ rather than $merge[\{log_arch\}]$ for sake of simplicity.

5 Log Analysis

In this section, we give the general method of our log analyzer. Given the values of the current logs and a particular claim, the aim is to establish the truthfulness of this claim based on the content of the logs.

5.1 Log Analyzer

First, we present the definition of the operation *PropAnalysis*.

```

scen, ok ← PropAnalysis(logs, prop) ≐
  PRE
    logs ∈ LOG_ARCH ∧ prop ∈ PROP ∧ agents(prop) ⊆ agents(logs)
  THEN
    scen := extractagents(prop)[merge[logs]];
    ok := scen ∩ val(prop)-1[{TRUE}]
  END
END
    
```

For a given set of logs *logs*, the operation *PropAnalysis* researches scenarios that hold for a given property *prop*. Two results are computed: the set of possible different scenarios (*scen*) zooming only on the concerned agents of the property, and among them, the subset of these scenarios that fulfill the property (*ok*). The ratio between the two sets *scen* and *ok* may inform us the level truthfulness of the researched property for the given logs.

Now, the method for the analysis of a claim of the form (*plain, def, prop*) proceeds as follows:

1. Select a set of logs in the global log architecture such that $\{plain, def\} \subseteq agents(prop)$. These *logs* may represent the current global set of logs or only a subset of them.
2. Execute $scen, ok \leftarrow PropAnalysis(logs, prop)$
3. Analyze the results in term of the admissibility of the claim and its explanation. For instance:
 - if $ok = scen$ the property holds for all scenarios, which leads us to conclude that the claim is valid and the defendant is responsible;
 - if $ok = \emptyset$ the property is false for all scenarios, and leads us to conclude that the claims should be rejected;
 - otherwise, an appropriate examination of scenarios that fulfill (or not) the property has to be conducted.

If the results are not conclusive, it could be necessary to increase the amount of logs used in the analysis, as we explain in Section 5.2.

Example 8. (claim *NoRoom* analysis) Suppose we have the following current logs representing the scenario illustrated in Figure 3:

$$\begin{aligned}
log_{Client} &= (\{Client\}, [(Send, Request), (Rec, Justify)]) \\
log_{Agency} &= (\{Agency\}, [(Rec, Request), (Send, Debit)]) \\
log_{Bank} &= (\{Bank\}, [(Rec, Debit), (Send, Justify)]) \\
log_{Hotel} &= (\{Hotel\}, [])
\end{aligned}$$

If we execute *PropAnalysis* for *NoRoom* using all these logs it is clear that *Agency* is liable for *Client*'s damages⁴, because only one scenario is generated and fulfills the property $prop_{NoRoom}$. However, suppose that obtaining log_{Bank} can be an issue and that we want to verify if *Agency* is responsible for the incident. Then, the property can be analyzed for a reduced architecture using only log_{Client} and log_{Agency} . This setting will generate the three scenarios described in Example 7, but for all of them $prop_{NoRoom}$ holds, which leads to the conclusion that the claim is valid and *Agency* should be responsible.

5.2 Incremental Analysis

When results are not precise enough, a deeper investigation can be conducted, for instance, inspecting more logs or in looking for some other dysfunctions. We focus now on an incremental approach to obtain more precise results when more logs are later added in the analysis.

Let $logs$ and $prop$ be respectively the logs and the property that have been already analyzed. Now let $logs'$ be a new set of logs, selected in order to obtain more precise results. Here we study how $scen', ok' \leftarrow PropAnalysis(logs \cup logs', prop)$ can be incrementally computed, reusing the results obtained by $scen, ok \leftarrow PropAnalysis(logs, prop)$.

Incremental calculus for $merge[logs \cup logs']$. The following property allows us to compute $merge[logs \cup logs']$.

Property 1. (merge by parts) Let $logs$ and $logs'$ be two sets of logs. We have:

$$merge[logs \cup logs'] = \text{union}(\{merge[logs' \cup \{log\}] \mid log \in merge[logs]\})$$

Then, in the first step of $PropAnalysis(logs \cup logs', prop)$ it is possible to reuse the result of $merge[logs]$ to compute $merge[logs \cup logs']$. However, the operation *extract* does not distribute on the operator *merge* (see Property 2 below).

Incremental calculus for $scen'$ and ok' . The results of $PropAnalysis(logs \cup logs', prop)$ can be approximated using the previous results for $scen$ and ok in the following way:

⁴ In law, the term “damage” is associated with an award of money to be paid as compensation for loss or injury. In this paper we refer to this term in a more general meaning.

```

iscen, iok ← IncrPropAnalysis(logs', prop, scen, ok) ≐
  PRE
    logs' ∈ LOG_ARCH ∧ prop ∈ PROP ∧
    scen ∈  $\mathcal{F}(\text{LOG\_FILE})$  ∧ ok ∈  $\mathcal{F}(\text{LOG\_FILE})$  ∧
    agents(prop) ⊆ agents(scen) ∧
    agents(prop) ⊆ agents(ok)
  THEN
    iscen := extractagents(prop)[union({merge[logs' ∪ {log]} | log ∈ scen})];
    iok := extractagents(prop)[union({merge[logs' ∪ {log]} | log ∈ ok})]
  END
END

```

Now we can compare:

$$\begin{aligned}
 \textit{scen}, \textit{ok} &\leftarrow \textit{PropAnalysis}(\textit{logs}, \textit{prop}) \\
 \textit{scen}', \textit{ok}' &\leftarrow \textit{PropAnalysis}(\textit{logs} \cup \textit{logs}', \textit{prop}) \\
 \textit{iscen}, \textit{iok} &\leftarrow \textit{IncrPropAnalysis}(\textit{logs}', \textit{prop}, \textit{scen}, \textit{ok})
 \end{aligned}$$

The result is:

$$\textit{ok}' \subseteq \textit{iok} \subseteq \textit{ok} \text{ and } \textit{scen}' \subseteq \textit{iscen} \subseteq \textit{scen}$$

This result can be concluded using the following property.

Property 2. (extraction by parts) Let *log* be a log, *log_arch* a log architecture, and *ags* a set of agents. We have:

$$\begin{aligned}
 &\textit{extract}_{\textit{ags}}[\textit{merge}[\textit{log_arch} \cup \{\textit{log}\}]] \subseteq \\
 &\textit{extract}_{\textit{ags}}[\textit{merge}[\textit{log_arch} \cup \{\textit{extract}_{\textit{ags}}(\textit{log})\}]]
 \end{aligned}$$

This property suggests that when an extraction is made before the merge some information about the order of events may be lost and may result in a large set of scenarios for *merge*. Since *IncrPropAnalysis* uses the results *scen* and *ok* of *PropAnalysis*, some information may be lost with the extraction of the events that only concern the agents of the property.

Application. Although we do not obtain exact results, this operation is interesting because it does not retest the researched property, what can be a complex step. Moreover in some cases we may obtain conclusive results. For instance, it is always the case where $\textit{iok} = \emptyset$ or $\textit{iok} = \textit{iscen}$.

Example 9. Suppose the claim $\textit{LateCancel} = (\textit{Client}, \textit{Agency}, \textit{prop}_{\textit{LateCancel}})$ such that:

$$\begin{aligned}
 &\textit{agents}(\textit{prop}_{\textit{LateCancel}}) = \{\textit{Client}, \textit{Agency}\} \wedge \\
 &\textit{val}(\textit{prop}_{\textit{LateCancel}}) = \lambda \textit{log}. (\textit{agents}(\textit{log}) = \{\textit{Client}, \textit{Agency}\} \mid \\
 &\quad (\textit{Send}, \textit{Cancel}) \in \textit{events}(\textit{log}) \wedge \\
 &\quad (\textit{Send}, \textit{Debit}) \in \textit{events}(\textit{log}) \wedge \\
 &\quad \textit{pos}((\textit{Send}, \textit{Cancel}), \textit{log}) < \textit{pos}((\textit{Send}, \textit{Debit}), \textit{log}))
 \end{aligned}$$

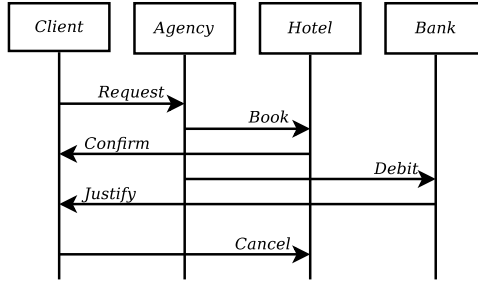


Fig. 4. Sequence diagram for possible scenario

That is, *Agency* is responsible for the claim *LateCancel* if *Client* sends a message to cancel the reservation before *Agency* sends the message to *Bank* charging the client. Now, consider the scenario described in Figure 4 and an initial call of *PropAnalysis* only with \log_{Client} and \log_{Agency} . The values for the logs can be written as follows:

$$\begin{aligned} \log_{Client} &= (\{Client\}, [(Send, Request), (Rec, Confirm), (Rec, Justify), \\ &\quad (Send, Cancel)]) \\ \log_{Agency} &= (\{Agency\}, [(Send, Request), (Send, Book), (Send, Debit)]) \end{aligned}$$

Then, it is not possible to verify if the claim is valid, because the result of $merge\{\{\log_{Client}, \log_{Agency}\}\}$ will produce 20 different scenarios where in ten of them the position of $Cancel_{Send}$ is before $Debit_{Send}$.

Now, we call *IncrPropAnalysis* with the result of the previous analysis and \log_{Bank} that should contain the following value:

$$\log_{Bank} = (\{Bank\}, [(Rec, Debit), (Send, Justify)])$$

The bank's log adds a restriction to the previous scenarios. Now we know that *Client* tries to cancel the reservation after the message $(Rec, Debit)$. After execute *IncrPropAnalysis* we remain with four scenarios in *iscen* because we are not sure about the position of the event $(Rec, Confirm)$. However, we obtain $iok = \emptyset$ and can conclude that the claim should be rejected.

6 Conclusion

This paper has presented some parts of the LISE context [15], a project with the objective to create a formal framework to precisely define liability in IT systems and establish liability in case of failure. Our framework provides a model for formally specifying the aspects of liability in a contractual setting. We also present the specification for a log analyzer that can be used to establish the validity of claims with relation to a given distributed log architecture. Finally, we explore the aspects of incremental analysis for this log analyzer.

6.1 Related Works

A large amount of work has been dedicated to the formal specification of contracts, among them [13, 9, 8]. Such specifications are useful but they still lack in specific issues of legal evidences and can be considered as a complementary approach to our framework.

Works in the domain of forensics [19, 20, 22, 26] and audit [24, 7, 27] make general references to analysis of digital information in a legal setting. However, in general these contributions seem to be targeted towards security issues and attack detection rather than define liability for possible claims that may rise.

The contributions presented in this paper differs from other works such as [25] and [7]. In these works the authors make reference to liability using logs but they are more focused in the aspects of monitorability of events and how the logs should be produced. The management of log distributions related with liability is an important related topic which has been covered by our prior work [16], where we characterized the acceptability of a given distribution with respect to a trust relationship and the neutrality of agents to log some given events.

6.2 Future Work

In this paper, we assumed that logs will not contain inconsistencies or incorrect values. As commented before, this hypothesis is justified by the use of other means, for instance digital signatures, that will assure characteristics such as integrity and non-repudiation for the logs. Another possibility, to verify log integrity, is the use of redundancy in a log architecture, when the events concerning one agent is recorded several times. However, situations where this hypothesis is not assumed has been considered in our previous works [16].

In the future we will extend our framework to take into account parameterized claims and properties (for instance to extend our case study with several clients and hotels). Future work also includes the integration of our work presented in [16] in this general framework.

Acknowledgement

This contribution is part of the LISE project (ANR-07-SESU-007) funded by ANR.

References

- [1] Abrial, J.: The B-Book. Cambridge University Press, Cambridge (1996)
- [2] Badeau, F., Amelot, A.: Using B as a High Level Programming Language in an Industrial Project: Roissy VAL. In: Treharne, H., King, S., Henson, M.C., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 334–354. Springer, Heidelberg (2005)
- [3] Barringer, H., Groce, A., Havelund, K., Smith, M.H.: An Entry Point for Formal Methods: Specification and Analysis of Event Logs. CoRR abs/1003.1682 (2010)

- [4] Barringer, H., Groce, A., Havelund, K., Smith, M.H.: Formal Analysis of Log Files. *Aerospace Computing, Information, and Communication* (to appear, 2010)
- [5] Behm, P., Benoit, P., Faivre, A., Meynadier, J.-M.: Météor: A Successful Application of B in a Large Project. In: Woodcock, J.C.P., Davies, J. (eds.) *FM 1999*. LNCS, vol. 1708, pp. 369–387. Springer, Heidelberg (1999)
- [6] Buskirk, E.V., Liu, V.T.: Digital Evidence: Challenging the Presumption of Reliability. *Journal of Digital Forensic Practice* 1(1), 19–26 (2006)
- [7] Cederquist, J.G., Corin, R., Dekker, M.A.C., Etalle, S., den Hartog, J.I., Lenzini, G.: Audit-based Compliance Control. *International Journal of Information Security* 6(2-3), 133–151 (2007)
- [8] Farrell, A.D.H., Sergot, M.J., Sallé, M., Bartolini, C.: Using the Event Calculus for Tracking the Normative State of Contracts. *International Journal of Cooperative Information Systems (IJCIS)* 14(2-3), 99–129 (2005)
- [9] Fenech, S., Pace, G.J., Schneider, G.: CLAN: A tool for contract analysis and conflict discovery. In: Liu, Z., Ravn, A.P. (eds.) *ATVA 2009*. LNCS, vol. 5799, pp. 90–96. Springer, Heidelberg (2009)
- [10] Grossi, D., Royakkers, L.M.M., Dignum, F.: Organizational Structure and Responsibility. *Artificial Intelligence and Law* 15(3), 223–249 (2007)
- [11] Hallal, H., Boroday, S., Petrenko, A., Ulrich, A.: A Formal Approach to Property Testing in Causally Consistent Distributed Traces. *Formal Aspects of Computing* 18(1), 63–83 (2006)
- [12] Insa, F.: The Admissibility of Electronic Evidence in Court (AEEC): Fighting against High-Tech Crime - Results of a European Study. *Journal of Digital Forensic Practice* 1(4), 285–289 (2006)
- [13] Kyas, M., Prisacariu, C., Schneider, G.: Run-Time Monitoring of Electronic Contracts. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) *ATVA 2008*. LNCS, vol. 5311, pp. 397–407. Springer, Heidelberg (2008)
- [14] Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM* 21(7), 558–565 (1978)
- [15] Le Métayer, D., Maarek, M., Mazza, E., Potet, M.L., Viet Triem Tong, V., Craipeau, N., Frénot, S., Hardouin, R.: Liability in Software Engineering: Overview of the LISE Approach and Illustration on a Case Study. In: *International Conference on Software Engineering (ICSE)*, pp. 135–144 (2010)
- [16] Le Métayer, D., Mazza, E., Potet, M.L.: Designing Log Architecture For Legal Evidence. In: *Software Engineering And Formal Methods (SEFM)*. IEEE, Los Alamitos (2010)
- [17] Lima, T.D., Royakkers, L.M.M., Dignum, F.: A Logic For Reasoning About Responsibility. *Logic Journal of the IGPL* 18(1), 99–117 (2010)
- [18] Maurer, U.M.: New Approaches to Digital Evidence. *Proceedings of the IEEE* 92(6), 933–947 (2004)
- [19] Peisert, S., Bishop, M., Karin, S., Marzullo, K.: Toward Models for Forensic Analysis. In: *Systematic Approaches to Digital Forensic Engineering*, pp. 3–15. IEEE, Los Alamitos (2007)
- [20] Rekhis, S., Krichène, J., Boudriga, N.: Cognitive-Maps Based Investigation of Digital Security Incidents. In: *Systematic Approaches to Digital Forensic Engineering*, pp. 25–40 (2008)
- [21] Saleh, M., Arasteh, A.R., Sakha, A., Debbabi, M.: Forensic Analysis of Logs: Modeling and verification. *Knowledge-Based Systems* 20(7), 671–682 (2007)
- [22] Sandler, D., Derr, K., Crosby, S.A., Wallach, D.S.: Finding the Evidence in Tamper-Evident Logs. In: *Systematic Approaches to Digital Forensic Engineering*, pp. 69–75 (2008)

- [23] Schneider, F.B.: Accountability for Perfection. *IEEE Security & Privacy* 7(2), 3–4 (2009)
- [24] Schneier, B., Kelsey, J.: Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security* 2(2), 159–176 (1999)
- [25] Skene, J., Raimondi, F., Emmerich, W.: Service-Level Agreements for Electronic Services. *IEEE Transactions on Software Engineering* 36(2), 288–304 (2010)
- [26] Stirewalt, R.E.K., Dillon, L.K., Kraemer, E.: The Inference Validity Problem in Legal Discovery. In: *International Conference on Software Engineering (ICSE)*, pp. 303–306. IEEE, Los Alamitos (2009)
- [27] Waters, B.R., Balfanz, D., Durfee, G., Smetters, D.K.: Building an Encrypted and Searchable Audit Log. In: *Proceedings of the Network and Distributed System Security*. The Internet Society (2004)

Formal Development of a Cardiac Pacemaker: From Specification to Code

Artur O. Gomes and Marcel V.M. Oliveira

Universidade Federal do Rio Grande do Norte, Brazil
artur.o.gomes@gmail.com, marcel@dimap.ufrn.br

Abstract. This paper presents a formal development of a cardiac pacing system based on a *Boston Scientific's* model, a pilot case study from the *Grand Challenge in Software Verification*. We present a summary of our Z model of the system, its translation into Perfect Developer, and the code generation and execution. Further practical result and analysis are also in the context of this paper.

Keywords: formal modelling, Z, refinement, Perfect Developer, pacemaker.

1 Introduction

The cardiac pacemaker [5], whose specification document was made available by the Software Quality Research Laboratory [1], at McMaster University in Canada, was proposed as one of the pilot case studies of the *Grand Challenges in Software Verification* [11], maintained by the *Verified Software Initiative (VSI)* [12]. The main objective of this project is to use formal methods throughout the development process of real case studies, stimulating researchers with experience in formal methods to apply their knowledge in a set of pilot case studies, like the pacemaker, showing empirically that formal methods have reached an acceptable level of usability even in industrial scale applications. The pacemaker is relevant since it is a very good example of a critical system which needs to have strong guarantees that the system will work according to its requirements. Any failure or bug may cause serious damages to patients.

In [9], we use the Z language [25] to formally specify the pacemaker. Moreover, we discuss how we used the theorem prover ProofPower-Z [17] to prove theorems that validate the consistency of the system. In this paper, we present our last results in the pacemaker case study by providing a means to execute the model: we present a translation of our Z model [9] into Perfect Developer [2]. Using this tool, we are able to verify the translated code by using its internal theorem prover fully automatically and refining the verified specification into programming languages like Java, C#, C++ or ADA.

Section 2 gives an overview of the pacemaker system. We will briefly present our model of the system using Z in Section 3. Then, we discuss some options that were considered to move into a verified code of the pacemaker in Section 4. In Section 5, we present some fragments of the translation of the pacemaker

Z specification into Perfect Developer. Finally, in Section 6 we discuss the results presented in this paper, followed by conclusions and future directions in Section 7

2 The Pacemaker

A pacemaker is a small electronic device that is capable of dealing with some of the human heart’s deficiencies. It is implanted into the human chest by surgery and connected to the heart via one or two thin leads [7]. Such leads are capable of pacing and sensing pulses from the heart and may be connected to the atrial chamber, to the ventricular chamber or to both chambers, depending on the heart’s needs. The pacemaker system [13] works basically in two modes: *permanent mode* running the main operation, *bradycardia therapy*, sensing and pacing pulses, and in *temporary mode* testing the pacemaker functionalities and emitting reports. During the *bradycardia therapy*, the pacemaker will be able to deliver pulses according to a set of parameters programmed by the cardiologist during the implant [22]. These parameters are related to the frequency of paced pulses, their voltages, and the type of response to sensed beats from the heart.

Among such parameters, the *bradycardia operation mode*, mentioned in our formalisation as *bo_mode*, is the parameter that describes how the therapy will behave regarding: (1) the chambers (atrium or ventricle) in which the pacemaker will sense pulses (chambers sensed); (2) the chambers to which the pacemaker will deliver pulses (chambers paced); (3) how the pacemaker will react to presence or absence of intrinsic pulses from the heart (response to sensing); and (4) whether or not, the pacemaker makes use of an accelerometer to increase the frequency of delivered pulses according to the intensity of body motion.

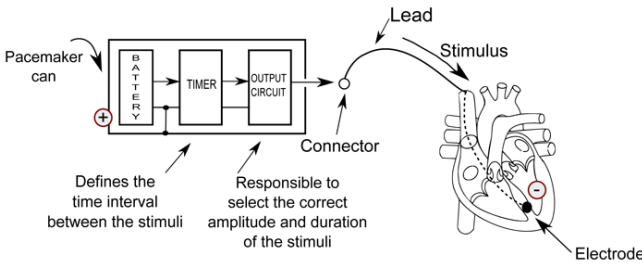


Fig. 1. VOO mode

As an example, the VOO mode, illustrated in Figure 1, is the *bradycardia operation mode* in which pulses are paced to the ventricle chamber (represented by the character 'V'), pulses are not sensed by the pacemaker (represented by the first character 'O'), and there is no response to sensing events (represented by the last character 'O'). In such mode, the pacemaker will deliver pulses to

the heart within an interval programmed by the cardiologist regardless of any electrical activity in the heart.

In the next section we present our formalisation of the pacemaker using Z.

3 Pacemaker in Z

Our formal specification of the pacemaker is based on an informal specification [13] of an old Boston Scientific's pacemaker model [5]. We decided to develop a model of the pacemaker, hereafter referred to as *pulse generator*, as a modularised Z state *PulseGenerator* making strong use of the Z schema calculus. It allows us to model each component of the pacemaker separately. The components of the *PulseGenerator* state are modelled as Z schemas with their variables and state invariants. The *PulseGenerator* [9] modularisation, as expected, helped us to keep the Z model comprehensible while moving into Perfect Developer. Furthermore, our specification also helped us in the development of a graphical user interface in C# because in the object oriented generated code each Z component was refined to a C# class, making it easier to access variables and methods from the *PulseGenerator* class.

Our *PulseGenerator* state is composed by a number of components like the set of programmable parameters, the set of measured parameters, the battery status component, and time. There are also many other components of the *PulseGenerator* state that are not presented here for the sake of conciseness¹. As an example, we briefly present the time state component, called *TimeSt*, which is responsible for storing all the information regarding time. It acts as a time counter and also stores information regarding the last occurrences of pulses from the heart, measured by the pacemaker.

<p><i>TimeSt</i></p> <p><i>time</i> : \mathbb{N}</p> <p><i>a_start_time</i>, <i>v_start_time</i>, <i>a_max</i>, <i>v_max</i> : \mathbb{N}</p> <p><i>a_delay</i>, <i>v_delay</i>, <i>a_curr_measurement</i>, <i>v_curr_measurement</i> : \mathbb{N}</p>

In our convention, variables related to the *atrial chamber* are prefixed with *a_* and, similarly, variables related to the *ventricular chamber* are prefixed with *v_*. Hence, the variable *a_start_time* stores the initial moment of a pulse detected in the atrium, *a_max* stores the maximum value of the pulse and *a_delay* stores the value of the duration of the pulse in the atrium chamber. The measurement of pulses coming from atrial and ventricle are defined as *a_curr_measurement* and *v_curr_measurement*, respectively.

The other components of the *PulseGenerator* state are modelled in a similar fashion. In some cases, however, state invariants are required for restricting the state components according to the informal specification. Finally, the *PulseGenerator* state is modelled as the conjunction of each component.

¹ The pacemaker full specification can be downloaded from <http://sites.google.com/site/pacemakerinz/pacemaker-in-z.pdf>

$$\begin{aligned}
 \text{PulseGenerator} \hat{=} & \text{PacingPulse} \wedge \text{SensingPulse} \wedge \text{TimeSt} \\
 & \wedge \text{MeasuredParameters} \wedge \text{Leads} \wedge \text{Accelerometer} \\
 & \wedge \text{EventMarkers} \wedge \text{BatteryStatus} \wedge \text{ImplantData} \\
 & \wedge \text{TelemetrySession} \wedge \text{ProgrammableParameters} \\
 & \wedge \text{MagnetTest}
 \end{aligned}$$

We model the pacemaker operations as Z operations over *PulseGenerator*. We illustrate our approach with the *SetTimer* operation below, which works as a counter for the pacemaker, incrementing the *time*. In Z, ΔSt denotes a change to the state *St*. We also state that $\theta(\text{PulseGenerator} \setminus (\text{time})) = \theta(\text{PulseGenerator} \setminus (\text{time}))'$, which means that only the *time* variable is changed; the remaining state variables are left unchanged.

$ \begin{aligned} & \text{SetTimer} \\ & \Delta \text{PulseGenerator} \\ & \text{time}' = \text{time} + 1 \\ & \theta(\text{PulseGenerator} \setminus (\text{time})) = \theta(\text{PulseGenerator} \setminus (\text{time}))' \end{aligned} $
--

According to its requirements [13], the pacemaker must be capable of sensing pulses from the heart by using leads connected to the heart’s chambers. There are three relevant moments during pulse measurement. As illustrated in Figure 2, the pacemaker must register the pulse *start* time, the *maximal* amplitude of the pulse and the moment when the pulse *ends*.

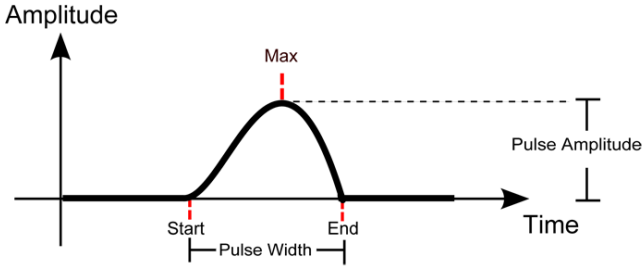


Fig. 2. Relevant moments during pulse measurement

As an example, we present the *VentricleStartTime* operation [7] that registers the initial moment of a sensed pulse in the ventricle chamber. Its precondition states that: (1) the current *bradycardia operation mode* must sense either the ventricle chamber or both chambers ($bo_mode.chambers_sensed \in \{C_VENTRICLE, C_DUAL\}$); (2) there was no previously sensed pulse in the ventricle ($r_wave = 0$); and (3) the measured pulse amplitude from ventricle chamber is higher than the actual pulse ($v_curr_measurement > r_wave$). If these preconditions are satisfied, the pacemaker will update the value of *r_wave*

with the value of $v_curr_measurement$ and register ($v_start_time' = time$) the initial time of that pulse.

$$\begin{array}{l}
 \hline
 \text{VentricleStartTime} \\
 \Delta\text{PulseGenerator} \\
 \hline
 bo_mode.chambers_sensed \in \{C_VENTRICLE, C_DUAL\} \\
 v_curr_measurement > r_wave \wedge r_wave = 0 \\
 r_wave' = v_curr_measurement \wedge v_start_time' = time \\
 \theta(\text{PulseGenerator} \setminus (r_wave, v_start_time))' \\
 = \theta(\text{PulseGenerator} \setminus (r_wave, v_start_time))
 \end{array}$$

We modelled the detection of pulses in the heart as distinct Z operations. As presented above, *VentricleStartTime* registers the initial moment of sensed pulse in the ventricle. A second operation, *VentricleMax*, is responsible for storing the maximal pulse amplitude. Finally, *VentricleEnd* registers the moment when the pulse ends. We also modelled the operation *VentricularMeasurement* as the disjunction of each of these operations as illustrated below. In a similar way, we modelled operations for the atrial chamber, which can be found in [8].

$$\text{VentricularMeasurement} \hat{=} \text{VentricleStart} \vee \text{VentricleMax} \vee \text{VentricleEnd}$$

The Z operation *SensingModule*, defined below, is responsible for the measurement of pulses coming from the heart. This operation is the disjunction of the *VentricularMeasurement* operation, presented above, and the operation *AtrialMeasurement* modelled similarly.

$$\text{SensingModule} \hat{=} \text{VentricularMeasurement} \vee \text{AtrialMeasurement}$$

The main purpose of the pacemaker is to supply the heart needs by delivering a therapy defined by the cardiologist. We specified the operation *BradyTherapy* (*bradycardia therapy*) as the sequential composition of four operations given below. The first one is the *SetTimer* presented above: the pacemaker increments its timer. Next, the operation *SensingModule*, as presented above, registers the exact moment of the pulses in the heart. Then, the pacemaker records the markers from sensed events, by invoking the operation *SensingMarkers*. Finally, *SetMode* (*set the bradycardia operation mode*), responds to the heart needs, based on the programmable parameters values defined by the cardiologist, and delivering pulses if necessary.

$$\text{BradTherapy} \hat{=} \text{SetTimer} \circledast \text{SensingModule} \circledast \text{SensingMarkers} \circledast \text{SetMode}$$

In [9], we discussed some of the verification made on this formal specification using the theorem prover ProofPower-Z. This verification included validating the system initialisation and its consistency regarding the state invariant. The next step, which is the subject of this paper, is to get a verified code of the pacemaker suitable to be executed. In the next section, we discuss some of the possibilities for moving from our Z specification into executable code.

4 Moving into Code

Before moving from the formal specification into the verified code, we ought to decide which methods we will use. A first way to proceed with the challenge is to run the resulting code in an 8-bit PIC micro controller made available by the Software Quality Research Laboratory, at the McMaster University (Canada). That board was specifically designed to support the pacemaker functionalities. However, due to budget restrictions we were not able to acquire that hardware from McMaster University. Nevertheless, we still have other possible solutions to simulate the verified code of the pacemaker. A first possible approach is to simulate the pacemaker on a *Field Programmable Gate Array* (FPGA). The strategy is to refine the Z specification to pseudo-code using the Z Refinement Calculus [3] possibly with tool support [16], and then, use the pseudo-code to write the program in languages like Handel-C [4] or SystemC [18], which are subsets of C and C++ with features for describing hardware. By adopting this solution, we would certainly spend a considerable amount of time refining the specification and discharging proof obligations. Another approach to reach source code is to translate our specification from Z to B using a tool like ProZ [19], an extension of ProB. As an advantage, B has notably very good tools for refinement and code generation such as Atelier-B and B-Toolkit, whose features are currently lacking for the Z language. However, it is very likely that the resulting specification would still require refinement within the B-Method. This would probably yield the need for intervention in order to discharge the proof obligations generated, which normally takes a reasonable amount of time. Furthermore, the modularisation of the Z specification might be compromised when translating into B.

Our approach is to translate the Z specification into Perfect Developer, a software produced by Escher Technologies, which provides fully automatic software verification and code generation to languages like C, C#, and ADA. The Perfect Developer language uses the notion of object orientation and can be easily learnt by Z users since it has a syntax similar to the Z language. One of the similarities between Z and Perfect Developer is the description of Z states in Perfect Developer using classes. Within the class, variables, invariants and initialisation are declared in separated sections. Moreover, schema operations in Z are translated into Perfect Developer in a similar fashion, except that *preconditions* and *postconditions* are explicitly stated in separated sections.

The support guaranteed by Escher Technologies with quick answers and software updates, allowing us to verify the entire specification of the pacemaker, was yet another motivation for choosing Perfect Developer. The tool also has a very good and automatic proof support. It automatically generates and discharges all proofs that are necessary to verify the specification in order to guarantee the consistency of the system. This means that Perfect Developer is able to verify, for example, whether the state invariant is satisfied and if the preconditions of each operation are satisfied. Some of these proofs have already been made in ProofPower-Z in previous work, as discussed in [9], which required a considerable amount of time. Hence, we have a tool that is capable of verifying our

formal specification with automatic proofs and also capable of a fully automatic refinement of that specification into a source code that meets its requirements. It means that we can avoid errors during the refinement process resulting in a code without errors fully automatically. In the the next section, we present the translation of our *Z* specification of the pacemaker to Perfect Developer.

5 From *Z* to Perfect Developer

The migration from *Z* into Perfect Developer is relatively straightforward due to their similarity. The main idea is to translate a *Z* state as a Perfect Developer class which will be refined to a Java or a C# class. A class in Perfect Developer has a basic structure: all the variables and its types are declared in the *abstract* section; the state invariants are declared in the *invariant*; in the *interface* we include all components like functions, schemas and properties which will be available (visible) to derived classes; in the *build* function we define which of the variables of the state will have their initial values declared, which will be refined to the constructor of the class in Java, for example. The *Z* schema operations over the state are modelled very similarly in Perfect Developer except that they are part of their related class. Moreover, the preconditions and postconditions of a schema operation are separated into *pre* and *post* in Perfect Developer, differently from *Z* where they are mixed in the same predicate. This normally requires a pre-condition calculation, which we have done using ProofPower-*Z*. However, despite being a complex system, the pacemaker pre-conditions were fairly simple to calculate.

The pacemaker proved to be an interesting challenge not only because of the complexity of its requirements, but also because of the complexity of its state and operations, which are modelled using a large number of variables and operations. In order to deal with this complexity, we adopted some methodologies and patterns while programming. This helped us to generate a more comprehensible code. Furthermore, a later formalisation of these patterns fosters the automation of such translation, which would allow a direct code generation from *Z* using Perfect Developer as a means to achieve that.

One such pattern is to create schema operations to update the values of the variables in a class, providing encapsulation of these variables. This makes it unnecessary to redeclare the same variable as an interface of each derived class. The pattern is as follows: for each variable included in the *interface* section, we create a schema operation, prefixed by *chg_* followed by the name of the variable. In each schema operation, we can change only the value of that variable. Moreover, unless the variables are included in the state invariants, there is no precondition for the schema.

For example, for the *TimeSt* *Z* component, we create a class in Perfect Developer named *TimeSt*, in which we include the same variables as presented in the *Z* model in the *abstract* section. In this case, there is no state invariant in the *TimeSt* *Z* component; hence, it is not necessary to include the *invariant* section. By omitting the *invariant* section, Perfect Developer will assume that there is no state invariant in the class.


```

class TimeSt ^=
  abstract
    var time : int;
    var a_start_time, a_max, a_delay : int;
    var v_start_time, v_max, v_delay : int;
    var a_curr_measurement, v_curr_measurement : int;
  interface
    function time, a_start_time, a_max, a_delay;
    function v_start_time, v_max, v_delay;
    function a_curr_measurement, v_curr_measurement;
  build{!time : int,
        !a_start_time : int, !a_max : int, !a_delay : int,
        !v_start_time : int, !v_max : int, !v_delay : int,
        !a_curr_measurement : int, !v_curr_measurement : int};
  schema !chg_time(x:int)
    post time! = x;
  ...
end;

```

The variables declared in the *abstract* section that are available to derived classes, must be included as *functions* in the *interface* section of the class. This is the case for all variables in the class *TimeSt*. We do not restrict any initial value for *TimeSt* variables, we only need to initiate the variables in the *build* function. This concludes the translation of the *TimeSt* component of the *PulseGenerator* into a Perfect Developer class.

A schema operation starts with the *schema* keyword, followed by the name of the schema prefixed by an exclamation mark '!'. When the operation has preconditions, they are included after the keyword *pre*. Finally, the postcondition of the operation is included after the keyword *post*, with each variable that is changed followed by an exclamation mark. For example, we have the *chg_time* schema presented above, in which only the variable *time* is updated.

Regarding the system composition, one of our main requirements whilst moving into Perfect Developer was to keep the modularisation of the system as in the Z specification. We were able to find a way to keep this modularisation in the translation of the *PulseGenerator* into Perfect Developer, by creating a new class *PulseGenerator* and instances of each component like *TimeSt* in the *abstract* section. In order to help us to identify which variables in *PulseGenerator* are declared as an instance of a component, we declare the variable as the name of the corresponding class, but prefixed by *c_*. For example, we declare an instance of *TimeSt* as *var c_TimeSt : TimeSt* in the main class *PulseGenerator*.

```

final class PulseGenerator ^=
  abstract
    var c_TimeSt : TimeSt;
    var c_ProgrammableParameters : ProgrammableParameters;
    var c_SensingPulse : SensingPulse;
    var c_PacingPulse : PacingPulse;
    var c_MeasuredParameters : MeasuredParameters;
    ...

```

Next, we have to declare all the variables of the class, those components of the state, as *function* in the *interface* section. It allows us to have access to its internal variables and schema operations. Finally, before translating the operations of the class, we have to declare the *build* function, which defines the initial values of the state. The variables declared in the *build* function will be translated to the parameters in the constructor of the class in languages like C#.

```
final class PulseGenerator ^=
...
interface
  function c_TimeSt, c_ProgrammableParameters, c_MeasuredParameters;
...
  function c_SensingPulse, function c_PacingPulse;
build{!c_TimeSt : TimeSt, !c_MeasuredParameters : MeasuredParameters,
!c_ProgrammableParameters : ProgrammableParameters,
!c_SensingPulse : SensingPulse, !c_PacingPulse : PacingPulse, ...};
...
```

The translation of the operations of the *PulseGenerator* to Perfect Developer is illustrated below, where we present the translation of the *SetTimer Z* operation. The syntax of an invocation of a component is a little bit different from that used to access a variable. In the *PulseGenerator* class, we access the variable *time*, for example, with the predicate *c_TimeSt.time*. However, we call an operation of the *c_TimeSt* component as *c_TimeSt!chg_time(...)*. Finally, we translate the *SetTimer* operation to a new schema operation of the class *PulseGenerator* that calls the operation *chg_time* and increments the actual time, *c_TimeSt.time*.

```
final class PulseGenerator ^=
...
  schema !SetTimer
    post (c_TimeSt!chg_time(c_TimeSt.time + 1));
...
```

As discussed above, a schema operation in Z is slightly different from its equivalent in Perfect Developer. Precondition and postcondition are separated in *pre* and *post* sections in Perfect Developer schemas. We use boolean functions to model the precondition of each schema operation in Perfect Developer. This approach allows the reuse of the precondition function every time we need to include the same precondition in an operation. In order to keep a comprehensible code, we established that a boolean function as the precondition of an operation is named with the prefix *pre*. First, we present the precondition of the *VentricleStartTime* operation.

```
function preVentricularStartTime : bool
  ^= (c_BO_MODE.bo_mode.chambers_sensed = CHAMBERS C_DUAL
      | c_BO_MODE.bo_mode.chambers_sensed = CHAMBERS C_VENTRICLE)
      & ((c_TimeSt.v_curr_measurement
          > c_MeasuredParameters.c_RWave.r_wave)
          & c_MeasuredParameters.c_RWave.r_wave = 0);
```

In this case, the precondition is modelled as the *preVentricleStartTime* function. We adapt some parts of the Z specification in the Perfect Developer syntax.

For instance, $bo_mode.chambers_sensed \in \{C_DUAL, C_VENTRICLE\}$ was translated to the following Perfect Developer predicate.

$$c_BO_MODE.bo_mode.chambers_sensed = CHAMBERS C_VENTRICLE \\ | c_BO_MODE.bo_mode.chambers_sensed = CHAMBERS C_DUAL$$

Using the precondition, we are able to translate the *VentricleStartTime* operation from Z to Perfect Developer, including the *preVentricleStartTime* function in the *pre* section, updating the initial time of the ventricular sensed pulse to the actual time ($c_TimeSt!chg_v_start_time(c_TimeSt.time)$) and updating the value of *r_wave* with the value of the current measurement in ventricle ($c_MeasuredParameters!set_r_wave(c_TimeSt.v_curr_measurement)$).

```
schema !VentricularStartTime
  pre preVentricularStartTime
  post (c_MeasuredParameters!set_r_wave(c_TimeSt.v_curr_measurement)
    & c_TimeSt!chg_v_start_time(c_TimeSt.time));
```

During the translation, we move from logical disjunctions of schemas in Z to guards in Perfect Developer. Those expressions are equivalent to 'if' and 'else' statements in languages like *C++* or *Java*. The basic syntax of a guard is [*precondition*] : *expression*. It means that for each guard, *if* the *precondition* is satisfied, a predicate *expression* will be the *postcondition* of the operation, *else*, the next guard will be checked. The operation will check the *precondition* of each guard until one is found to be *true*. If none of the guards are *true*, the last guard is selected. We include closed brackets '[]' with no preconditions as the last guard, which means, in Perfect Developer, that if none of the preconditions are satisfied, the operation will be skipped. Note that there is no global preconditions in the *SensingModule* operation: the preconditions modelled as boolean functions like *preVentricularMeasurement* are included inside the brackets in each guard in the 'post' section of the operation.

```
schema !SensingModule
  post ([preVentricularMeasurement]: !VentricularMeasurement,
    [preAtrialMeasurement]: !AtrialMeasurement,
    []);
```

Finally, Z sequential composition is translated into Perfect Developer's *then*-constructs. Besides, each schema operation is called using an exclamation mark '!', as illustrated below, where we present the translation of the *BradyTherapy* operation, presented in Section 3.

```
schema !BradyTherapy
  post (!SetTimer then !SensingModule
    then !SensingMarkers then !SetMode);
end;
```

In this section, we presented a brief overview of the translation from the Z specification of the pacemaker into Perfect Developer. Based on this translation, we were able to automatically generate C# code and execute a simulation of the pacemaker using a graphical user interface. In the next section, we provide an analysis of the results we have achieved.

6 Analysis of Results

Our decision to adopt Perfect Developer was based on the similarities to the Z language like the use of state invariants and operations as schemas. Furthermore, the possibility to automatically generate verified code was also an attractive factor in our choice. Throughout the development process we had a very good support from the software vendors, who helped us to solve issues during the translation to the new syntax and even removed some of the limitations of the software's free version.

The translation from Z to Perfect Developer required some adaptations of our specification to the syntax of the latter. Some of those adaptations have been presented in this current paper, like the use of classes to model components and the *PulseGenerator* itself, and the instantiation of each component as a member of the system state. Besides, we had to explicitly state the preconditions and postconditions that are not separated in Z. We have also translated the remaining operations, those related to the tests made by the pacemaker, where it generates reports of events occurred during the therapy.

In a first version of the translation into Perfect Developer, we generated over 3360 lines of code; the *PulseGenerator* class alone contained 1300 lines of code. The generation resulted in over 740 proof obligations that were automatically proved by the Perfect Developer theorem prover. Using an *Intel Core2 Duo 2.4 Ghz* with 3Gb DDR2 RAM machine, all proofs were discharged in less than 5 minutes. This version reached 116% of Perfect Developers Academic license capacity, which is based on the number of source code tokens parsed. For this reason, a specific version was provided to us by Escher Technologies, which allowed us to conclude the whole development.

Following a suggestion from the Perfect Developer support team, we developed a second version of the system. Based on their suggestion, we made some modifications in the code by moving the preconditions of the schema operations of *PulseGenerator* class to boolean functions, allowing their reuse in other operations. For instance, the *PulseGenerator* class was reduced from 1300 to 750 lines of code. As a consequence, the number of proof obligations was also reduced to around 560. The overall time spent for discharging proofs was 86 seconds, since most of the proofs required less than one second to be discharged. Thus, this change considerably reduced the size of the generated executable code to an amount of 9585 lines of code. This version reached almost 65% of Perfect Developers Academic license capacity.

The work presented in this paper is a significant development of the work presented in [9]. First, we were able to verify the entire version of our Z model

translated into Perfect Developer. We also automatically generated C# code from the verified specification in less than five seconds. The refinement resulted in over 9000 lines of verified C# code². The choice of C# was due to the fact that the interface between hand-written code and the code generated by Perfect Developer is easier for C# than for other languages. Besides, C# provides tools that makes it faster to develop a GUI for the pacemaker system. In Figure 3 we illustrate our graphical user interface of the pacemaker developed using C# under *Microsoft Visual C# 2010*.

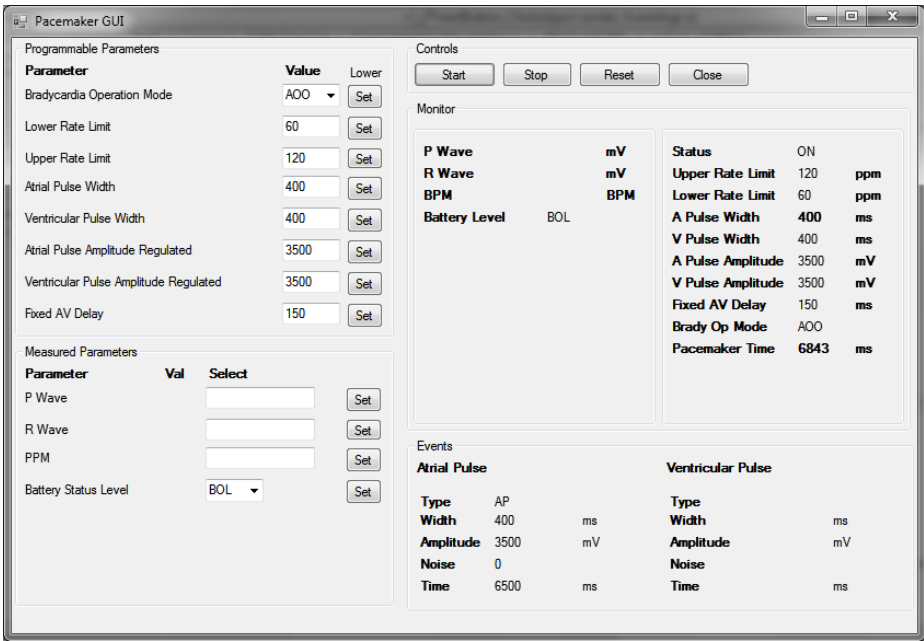


Fig. 3. Pacemaker GUI using C#

According to its requirements [13], the pacemaker must be able to print reports of the events occurred during the therapy. We modelled those reports in Z by creating schema operations using output variables (those followed by an exclamation mark '!'). As an example, we present the operation *OutputActualTime*, which simply outputs the actual value of the time in the *PulseGenerator* state. For output operations, we include the predicate \exists *PulseGenerator*, instead of Δ *PulseGenerator*, to assure that the operation does not change the state of the system.

² Available at <http://sites.google.com/site/pacemakerinz/pacemaker-in-csharp.zip>

$\frac{\text{OutputActualTime}}{\exists \text{PulseGenerator}}$	$\text{out_time!} : \text{int}$
	$\text{out_time!} = \text{time}$

We were not able to translate those output operations to Perfect Developer because we cannot describe precisely what is the effect of the output operation in the executable code. For example, there is no way to precisely define whether the output will be an output stream or a call to the operating system. The solution is to translate those operations directly to the executable code, in our case, C#, by creating an instance of the *PulseGenerator* class in which we develop those methods equivalent to the output operations modelled in Z. In this way, we can have those output operations, translated into C#, printing reports directly in the graphical user interface developed by us.

7 Conclusions

In this paper we present our recent results in the formal development of a cardiac pacemaker, one of the challenges in software verification suggested by the Verified Software Initiative. Here, we provide a step forward from [9], where we present our Z model and its verification using ProofPower-Z. In this paper, we describe the approach used to encode the cardiac pacemaker based on the previously presented Z model.

In a later stage, some possibilities for encoding the Z model have been considered and discussed in Section 4. A first possibility is to refine the Z model using the Z Refinement Calculus. Although this would have a mechanical support, a large amount of time would still be needed to discharge the proof obligations. Another possibility is to translate the code into B by using ProB and then use a B tool like Atelier-B to refine the specification to code mechanically. This approach, however, would still need some intervention since it would require further refinement of the B model resulting from the translation. Finally, we discussed the possibility to translate the Z model into Perfect Developer in order to get an automatic code generation of the verified specification.

We have analysed the tool support for the refinement of the Z specification into code and concluded that Perfect Developer could be a good way for this purpose, since it provides an automatic generation of verified code and has an excellent professional support team. This approach proved to be much easier than refining the Z specification using the Z refinement calculus, as mentioned in [9], which lacks stable tool support yielding to a need for handcrafted refinement and proofs. However, we do not provide a formalisation and proof of the translation from Z into Perfect Developer, which is left as an interesting piece of future work. Currently, we rely on the fact that the translation from Z into Perfect Developer is fairly direct.

This case study proved to be a complex task since the informal specification [13] does not provide enough information for a software development team

to construct a pacemaker system without any previous knowledge on basic cardiological information such as *pacing modes, timing cycles, and event markers*. Researchers that intend to embrace the challenge need a large amount of additional resources such as pacemaker guides [22] and books in the cardiology domain [7]. As a consequence of the lack of a detailed specification of some components originally presented in [13] due to commercial reasons, a small number of functionalities of the pacemaker have not yet been modelled. For instance, we have no specific information about the internal operations of the *rate-adaptive algorithm*, which is used by the pacemaker to recognise the level of body's movement and increase the frequency of pulses delivered. Besides *time*, we were not able to specify other non-functional requirements due to the lack of specification in the original document.

There are other research groups that also have undertaken the pacemaker case study. An informal one-day meeting was held in November 2009, during the *Second World Congress on Formal Methods* in Eindhoven. Research groups involved with the pacemaker case study, including researchers from Boston Scientific, were present in this meeting. On this occasion, the research groups could present their most recent results on the pacemaker challenge and compare their work. Moreover, we could discuss some issues related to the challenge, such as the lack of a detailed specification of some operations of the pacemaker, as discussed in this paper, and also the availability of possible test scenarios to be used in order to validate the pacemaker system. Despite the widespread interest in the challenge, few results have been published.

An approach using VDM was used by Macedo *et. al* in [14], where they constructed a model based on a subset of the functionalities of the pacemaker in VDM. They modelled 8 of the 19 bradycardia operation modes including the corresponding programmable parameters and developed not only a sequential model of the system, but also a *concurrent* and a *distributed real-time* model of the pacemaker using VDM++. They constructed several test cases in order to validate their sequential model, and re-used some of these to validate their *concurrent* and also their *distributed real-time* model.

Event-B [15] has been used by researchers from INRIA to undertake the pacemaker case study. This work suggests an incremental approach in which new functionalities of the system are added to the model at each refinement step. They validate their system using ProB with many test scenarios like *absence of intrinsic pulses from the heart* and *rate-adaptive pacing modes*.

Finally, in [24], Tuan *et. al* proposed a formal model of the pacemaker based on its behaviour including the communication with the external environment. They created a real-time model of the pacemaker using timed extensions of CSP [20] and used the model checker *Process Analysis Toolkit (PAT)* [23] in order to verify critical properties, such as *deadlock freeness* and *heart rate limits*.

In a near future, we will investigate approaches to generate hardware prototypes from our Perfect Developer code. Furthermore, we also intend to specify the missing parts of the specification as soon as we receive a more detailed information regarding those features, from Boston Scientific and the Software

Quality Research Laboratory at McMaster University. Another interesting piece of future work is to create some test scenarios to validate our system.

In the long term agenda, we intend to formalise the translation from *Z* to Perfect Developer. Besides guaranteeing the correction of the translation presented in this paper, this formalisation can be used as a guideline in the implementation of a tool that mechanises the translation. Moreover, a test suite could be used in order to improve confidence of the translation. Yet another piece of interesting work for the long term is to extend our model to include concurrency using *Circus* [6], a combination of *Z* and CSP, and the derivation of the *Circus* specification into code using its refinement calculus and tool support [10]. Notions of real-time are also to be incorporated in our model in a later stage using *Circus*' timed version [21].

Acknowledgments

Jim Woodcock originally proposed this challenge. David Crocker gave an very good support and kindly reduced the limitations of the Perfect Developer free edition. INES and CNPq partially supports the work of the authors: grants 550946/2007-1, 620132/2008-6, 573964/2008-4, and 476836/2009-3.

References

1. Software Quality Research Laboratory SQRL (2008), <http://www.cas.mcmaster.ca/sqrl/>
2. Carter, G., Monahan, R., Morris, J.M.: Software Refinement With Perfect Developer. In: SEFM 2005: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods, Washington, DC, USA, pp. 363–373. IEEE Computer Society, Los Alamitos (2005)
3. Cavalcanti, A., Woodcock, J.: ZRC - A Refinement Calculus for Z. *Formal Aspects of Computing* 10(3), 267–289 (1998)
4. Celoxica. Handel-C language reference manual, v3.0 (2002)
5. Boston Scientific Corporation. ALTRUA Pacemaker System Guide (2008)
6. de Oliveira, M.V.M.: Formal Derivation of State-Rich Reactive Programs Using Circus PhD thesis, Department of Computer Science, University of York (2005) YCST-2006/02
7. Ellenbogen, K.A., Wood, M.A.: Cardiac Pacemakers and ICDs. Wiley-Blackwell (2005)
8. Gomes, A.O., de Oliveira, M.V.M.: Pacemaker Specification in Z Using ProofPower-Z
9. Gomes, A.O., de Oliveira, M.V.M.: Formal Specification of a Cardiac Pacing System. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 692–707. Springer, Heidelberg (2009)
10. Gurgel, A.C., Castro, C.G., de Oliveira, M.V.M.: Tool Support for the Circus Refinement Calculus. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, p. 349. Springer, Heidelberg (2008)
11. Hoare, T.: The Verifying Compiler: A Grand Challenge for Computing Research. *Journal of the ACM* 50 (2003)

12. Hoare, T., Leavens, G.T., Misra, J., Shankar, N.: *The Verified Software Initiative: A Manifesto* (2007)
13. Software Quality Research Laboratory. *Pacemaker System Specification* (2007), http://sqr1.mcmaster.ca/_SQRLDocuments/PACEMAKER.pdf
14. Macedo, H.D., Larsen, P.G., Fitzgerald, J.S.: Incremental Development of a Distributed Real-Time Model of a Cardiac Pacing System Using VDM. In: Cuéllar, J., Maibaum, T.S.E., Sere, K. (eds.) *FM 2008*. LNCS, vol. 5014, pp. 181–197. Springer, Heidelberg (2008)
15. Méry, D., Singh, N.K.: *Pacemaker's Functional Behaviors in Event-B*. Technical Report Version 2, Universit Henri Poincar Nancy 1 (2009)
16. Oliveira, M.V.M., Gurgel, A.C., Castro, C.G.: CRefine: Support for the Circus Refinement Calculus. In: *SEFM 2008: Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, Washington, DC, USA, 2008, pp. 281–290. IEEE Computer Society, Los Alamitos (2008)
17. Oliveira, M., Cavalcanti, A., Woodcock, J.: *Unifying Theories in ProofPower-Z*. Formal Aspects of Computing (2007)
18. Panda, P.R.: *SystemC: A Modeling Platform Supporting Multiple Design Abstractions*. In: *ISSS 2001: Proceedings of the 14th International Symposium on Systems Synthesis*, pp. 75–80. ACM, New York (2001)
19. Plagge, D., Leuschel, M.: Validating Z Specifications using the ProB Animator and Model Checker. In: Davies, J., Gibbons, J. (eds.) *IFM 2007*. LNCS, vol. 4591, pp. 480–500. Springer, Heidelberg (2007)
20. Schneider, S.: *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, Inc., New York (1999)
21. Sherif, A.: *A Framework for Specification and Validation of Real-Time Systems using Circus Actions*. PhD thesis, Center of Informatics - Federal University of Pernambuco, Brazil (2006)
22. Stroobandt, R., Barold, A.F.S.S.: *Cardiac Pacemakers Step by Step – An Illustrated Guide*. Blackwell Publishing Ltd, Malden (2003)
23. Sun, J., Liu, Y., Dong, J.S.: Model Checking CSP Revisited: Introducing a Process Analysis Toolkit. In: Margaria, T., Steffen, B. (eds.) *ISoLA. Communications in Computer and Information Science*, vol. 17, pp. 307–322. Springer, Heidelberg (2008)
24. Tuan, L.A., Zheng, M.C., Tho, Q.T.: Modeling and Verification of Safety Critical Systems: A Case Study on Pacemaker. In: *Fourth IEEE International Conference on Secure Software Integration and Reliability Improvement*. IEEE Press, Los Alamitos (2010)
25. Woodcock, J.C.P., Davies, J.: *Using Z-Specification, Refinement, and Proof*. Prentice-Hall, Englewood Cliffs (1996)

A Decision Procedure for Bisimilarity of Generalized Regular Expressions

Marcello Bonsangue¹, Georgiana Caltais^{2,3}, Eugen-Ioan Goriac^{2,3},
Dorel Lucanu³, Jan Rutten^{4,5,6}, and Alexandra Silva⁴

¹ LIACS - Leiden University, The Netherlands
marcello@liacs.nl

² School of Computer Science - Reykjavik University, Iceland
{gcaltais10,egoriac10}@ru.is

³ Faculty of Computer Science - Alexandru Ioan Cuza University, Romania

⁴ Centrum voor Wiskunde en Informatica, The Netherlands
dlucanu@info.uaic.ro

⁵ Radboud University Nijmegen, The Netherlands
{janr,ams}@cwi.nl

⁶ Vrije Universiteit Amsterdam, The Netherlands

Abstract. A notion of generalized regular expressions for a large class of systems modeled as coalgebras, and an analogue of Kleene’s theorem and Kleene algebra, were recently proposed by a subset of the authors of this paper. Examples of the systems covered include infinite streams, deterministic automata and Mealy machines. In this paper, we present a novel algorithm and a tool to decide whether two expressions are bisimilar or not. The procedure is implemented in the automatic theorem prover CIRC, by reducing coinduction to an entailment relation between an algebraic specification and an appropriate set of equations.

1 Introduction

Regular expressions and deterministic automata (DFA’s) constitute two of the most basic structures in computer science. Kleene’s theorem [8] gives a fundamental correspondence between these two structures: each regular expression denotes a language that can be recognized by a DFA and, conversely, the language accepted by a DFA can be specified by a regular expression. Languages denoted by regular expressions are called regular. Two regular expressions are (language) equivalent if they denote the same regular language. Salomaa [14] presented a sound and complete axiomatization (later refined by Kozen in [9,10]) for proving the equivalence of regular expressions.

Coalgebras arose in the last decade as a suitable mathematical framework to study state-based systems, such as DFA’s. For a functor $\mathcal{G}: \mathbf{Set} \rightarrow \mathbf{Set}$, a \mathcal{G} -coalgebra or \mathcal{G} -system is a pair (S, g) , consisting of a set S of states and a function $g: S \rightarrow \mathcal{G}(S)$ defining the “transitions” of the states. We call the functor \mathcal{G} the type of the system. For instance, DFA’s can be readily modeled as finite coalgebras of the functor $\mathcal{G}(S) = 2 \times S^A$.

For coalgebras of a large class of functors, a language of regular expressions; a corresponding generalization of Kleene's theorem; and a sound and complete axiomatization for the associated notion of behavioral equivalence were introduced in [21]. Both the language of expressions and their axiomatization were derived, in a modular fashion, from the functor defining the type of the system.

Algebra and related tools can be successfully used for reasoning on properties of systems. In this paper, we present a novel method for checking for the bisimilarity of generalized regular expressions using the coinductive theorem prover CIRC [412]. The main novelty of the method lies on the generality of the systems it can handle. CIRC is a metalanguage application implemented in Maude [3], and its target is to prove properties over infinite data structures. It has been successfully used for checking the equivalence of programs, and trace equivalence and strong bisimilarity of processes. The tool may be tested online and downloaded from <http://fsl.cs.uiuc.edu/index.php/Circ>.

The main contributions of this paper can be summarized as follows. We present the algebraic counterpart of the coalgebraic framework of the generalized regular expressions mentioned above. This enables us to automatically derive algebraic specifications that model the language of expressions, and to define an appropriate equational entailment relation for checking for the behavioural equivalence of expressions. Furthermore, the implementation of both the algebraic specification and the entailment relation in CIRC allows for automatic reasoning on the equivalence of expressions.

Organization of the paper. Section 2 recalls the basic definitions of the language associated to a polynomial functor. Section 3 formulates the aforementioned language as an algebraic specification, which paves the way to implement in CIRC a procedure to decide equivalence of expressions. The decision procedure and the soundness of its implementation in CIRC are described in Section 4. In Section 4.1 we show, by means of examples, how one can check for bisimilarity, using CIRC. Section 5 contains concluding remarks and pointers for future work.

2 Regular Expressions for Polynomial Coalgebras

In this section, we briefly recall the basic definitions in [215].

Let **Set** denote the category of sets (represented by capital letters X, Y, \dots) and functions (represented by lower case letters f, g, \dots). The notation Y^X represents the family of functions from X to Y . The product of two sets X, Y is written as $X \times Y$ and has the projections functions π_1 and π_2 : $X \xleftarrow{\pi_1} X \times Y \xrightarrow{\pi_2} Y$. We define $X \diamond Y = X \uplus Y \uplus \{\perp, \top\}$ where \uplus is the disjoint union of sets, with injections $X \xrightarrow{\kappa_1} X \uplus Y \xleftarrow{\kappa_2} Y$. Note that the set $X \diamond Y$ is different from the classical coproduct of X and Y (which we shall denote by $X + Y$), because of the two extra elements \perp and \top . These extra elements will later be used to represent, respectively, underspecification and inconsistency in the specification of some systems.

For each of the operations defined above on sets, there are analogous ones on functions. Let $f: X \rightarrow Y$, $f_1: X \rightarrow Y$ and $f_2: Z \rightarrow W$. We define the following operations:

$$\begin{aligned}
 f_1 \times f_2: X \times Z &\rightarrow Y \times W & f_1 \diamond f_2: X \diamond Z &\rightarrow Y \diamond W \\
 (f_1 \times f_2)(\langle x, z \rangle) &= \langle f_1(x), f_2(z) \rangle & (f_1 \diamond f_2)(c) &= c, \quad c \in \{\perp, \top\} \\
 f^A: X^A &\rightarrow Y^A & (f_1 \diamond f_2)(\kappa_i(x)) &= \kappa_i(f_i(x)), \quad i \in \{1, 2\} \\
 f^A(g) &= f \circ g & &
 \end{aligned}$$

Note that here we are using the same symbols that we defined above for the operations on sets. It will always be clear from the context which operation is being used.

In our definition of non-deterministic functors we will use constant sets equipped with an information order. In particular, we will use join-semilattices. A (bounded) join-semilattice is a set \mathbf{B} equipped with a binary operation $\vee_{\mathbf{B}}$ and a constant $\perp_{\mathbf{B}} \in \mathbf{B}$, such that $\vee_{\mathbf{B}}$ is commutative, associative and idempotent. The element $\perp_{\mathbf{B}}$ is neutral with respect to $\vee_{\mathbf{B}}$. As usual, $\vee_{\mathbf{B}}$ gives rise to a partial ordering $\leq_{\mathbf{B}}$ on the elements of \mathbf{B} : $b_1 \leq_{\mathbf{B}} b_2 \Leftrightarrow b_1 \vee_{\mathbf{B}} b_2 = b_2$. Every set S can be mapped into a join-semilattice by taking \mathbf{B} to be the set of all finite subsets of S with union as join.

Coalgebras. A coalgebra is a pair $(S, g: S \rightarrow \mathcal{G}(S))$, where S is a set of states and $\mathcal{G}: \mathbf{Set} \rightarrow \mathbf{Set}$ is a functor. The functor \mathcal{G} , together with the function g , determines the *transition structure* (or dynamics) of the \mathcal{G} -coalgebra [13].

Definition 1 (Bisimulation). Let (S, f) and (T, g) be two \mathcal{G} -coalgebras. We call a relation $R \subseteq S \times T$ a bisimulation [7] iff

$$(s, t) \in R \Rightarrow \langle f(s), g(t) \rangle \in \overline{\mathcal{G}}(R)$$

where $\overline{\mathcal{G}}(R)$ is defined as $\overline{\mathcal{G}}(R) = \{\langle \mathcal{G}(\pi_1)(x), \mathcal{G}(\pi_2)(x) \rangle \mid x \in \mathcal{G}(R)\}$.

We write $s \sim_{\mathcal{G}} t$ whenever there exists a bisimulation relation containing (s, t) and we call $\sim_{\mathcal{G}}$ the bisimilarity relation. We shall drop the subscript \mathcal{G} whenever the functor \mathcal{G} is clear from the context.

Polynomial functors. They are functors $\mathcal{G}: \mathbf{Set} \rightarrow \mathbf{Set}$, built inductively from the identity, and constants, using \times , \diamond and $(-)^A$:

$$PF \ni \mathcal{G} ::= \text{Id} \mid \mathbf{B} \mid \mathcal{G} \diamond \mathcal{G} \mid \mathcal{G} \times \mathcal{G} \mid \mathcal{G}^A \tag{1}$$

where \mathbf{B} is a (non-empty) finite join-semilattice and A is a finite set. Typical examples of polynomial functors include $\mathcal{R} = \mathbf{B} \times \text{Id}$, $\mathcal{M} = (\mathbf{B} \times \text{Id})^A$, $\mathcal{D} = 2 \times \text{Id}^A$ and $\mathcal{Q} = (1 \diamond \text{Id})^A$. These functors represent, respectively, the type of Mealy, deterministic and partial deterministic automata. \mathcal{R} -bisimulation is stream equality, whereas \mathcal{D} -bisimulation coincides with language equivalence.

Next, we give the definition of the ingredient relation, which relates a polynomial functor \mathcal{G} with its *ingredients*, i.e. the functors used in its inductive construction. We shall use this relation later for typing our expressions.

Definition 2. Let $\triangleleft \subseteq PF \times PF$ be the least reflexive and transitive relation on polynomial functors such that

$$\mathcal{G}_1 \triangleleft \mathcal{G}_1 \times \mathcal{G}_2, \quad \mathcal{G}_2 \triangleleft \mathcal{G}_1 \times \mathcal{G}_2, \quad \mathcal{G}_1 \triangleleft \mathcal{G}_1 \diamond \mathcal{G}_2, \quad \mathcal{G}_2 \triangleleft \mathcal{G}_1 \diamond \mathcal{G}_2, \quad \mathcal{G} \triangleleft \mathcal{G}^A$$

Here and throughout this document we use $\mathcal{F} \triangleleft \mathcal{G}$ as a shorthand for $\langle \mathcal{F}, \mathcal{G} \rangle \in \triangleleft$. If $\mathcal{F} \triangleleft \mathcal{G}$, then \mathcal{F} is said to be an *ingredient* of \mathcal{G} . For example, 2 , ld , ld^A and \mathcal{D} itself are all the ingredients of the deterministic automata functor \mathcal{D} .

A language of regular expressions for polynomial coalgebras. We now associate a language of expressions $\text{Exp}_{\mathcal{G}}$ with each polynomial functor \mathcal{G} .

Definition 3 (Expressions). Let A be a finite set, \mathbf{B} a finite join-semilattice and X a set of fixed-point variables. The set Exp of all expressions is given by the following grammar, where $a \in A$, $b \in \mathbf{B}$ and $x \in X$:

$$\varepsilon ::= \emptyset \mid x \mid \varepsilon \oplus \varepsilon \mid \mu x. \gamma \mid b \mid l\langle \varepsilon \rangle \mid r\langle \varepsilon \rangle \mid l[\varepsilon] \mid r[\varepsilon] \mid a(\varepsilon) \quad (2)$$

where γ is a guarded expression given by:

$$\gamma ::= \emptyset \mid \gamma \oplus \gamma \mid \mu x. \gamma \mid b \mid l\langle \varepsilon \rangle \mid r\langle \varepsilon \rangle \mid l[\varepsilon] \mid r[\varepsilon] \mid a(\varepsilon) \quad (3)$$

In the expression $\mu x. \gamma$, μ is a binder for all the free occurrences of x in γ . Variables that are not bound are free. A *closed expression* is an expression without free occurrences of fixed-point variables x . We denote the set of closed expressions by Exp^c .

The language of expressions for polynomial coalgebras is a generalization of the classical notion of regular expressions: \emptyset , $\varepsilon_1 \oplus \varepsilon_2$ and $\mu x. \gamma$ play similar roles to the regular expressions denoting empty language, the union of languages and the Kleene star. The expressions $l\langle \varepsilon \rangle$, $r\langle \varepsilon \rangle$, $l[\varepsilon]$, $r[\varepsilon]$ and $a(\varepsilon)$ refer to the left and right hand-side of products and coproducts, and function application, respectively. Next, we present a type assignment system for associating expressions to polynomial functors. This will allow us to associate with each functor \mathcal{G} the expressions $\varepsilon \in \text{Exp}^c$ that are valid specifications of \mathcal{G} -coalgebras.

Definition 4 (Type system). We now define a typing relation $\vdash \subseteq \text{Exp} \times PF \times PF$ that will associate an expression ε with two polynomial functors \mathcal{F} and \mathcal{G} , which are related by the ingredient relation (\mathcal{F} is an ingredient of \mathcal{G}). We shall write $\vdash \varepsilon: \mathcal{F} \triangleleft \mathcal{G}$ for $\langle \varepsilon, \mathcal{F}, \mathcal{G} \rangle \in \vdash$. The rules that define \vdash are the following:

$$\begin{array}{c}
\frac{}{\vdash \emptyset: \mathcal{F} \triangleleft \mathcal{G}} \quad \frac{}{\vdash b: \mathbf{B} \triangleleft \mathcal{G}} \quad \frac{}{\vdash x: \mathcal{G} \triangleleft \mathcal{G}} \quad \frac{\vdash \varepsilon: \mathcal{G} \triangleleft \mathcal{G}}{\vdash \mu x. \varepsilon: \mathcal{G} \triangleleft \mathcal{G}} \\
\frac{\vdash \varepsilon_1: \mathcal{F} \triangleleft \mathcal{G} \quad \vdash \varepsilon_2: \mathcal{F} \triangleleft \mathcal{G}}{\vdash \varepsilon_1 \oplus \varepsilon_2: \mathcal{F} \triangleleft \mathcal{G}} \quad \frac{}{\vdash \varepsilon: \text{ld} \triangleleft \mathcal{G}} \quad \frac{}{\vdash r[\varepsilon]: \mathcal{F}_1 \diamond \mathcal{F}_2 \triangleleft \mathcal{G}} \quad \frac{}{\vdash a(\varepsilon): \mathcal{F}^A \triangleleft \mathcal{G}} \\
\frac{}{\vdash \varepsilon: \mathcal{F}_1 \triangleleft \mathcal{G}} \quad \frac{}{\vdash \varepsilon: \mathcal{F}_2 \triangleleft \mathcal{G}} \quad \frac{}{\vdash \varepsilon: \mathcal{F}_1 \triangleleft \mathcal{G}} \\
\frac{}{\vdash l\langle \varepsilon \rangle: \mathcal{F}_1 \times \mathcal{F}_2 \triangleleft \mathcal{G}} \quad \frac{}{\vdash r\langle \varepsilon \rangle: \mathcal{F}_1 \times \mathcal{F}_2 \triangleleft \mathcal{G}} \quad \frac{}{\vdash l[\varepsilon]: \mathcal{F}_1 \diamond \mathcal{F}_2 \triangleleft \mathcal{G}}
\end{array}$$

We can now formally define the set of \mathcal{G} -expressions: well-typed expressions associated with a polynomial functor \mathcal{G} .

Definition 5 (\mathcal{G} -expressions). *Let \mathcal{G} be a polynomial functor and \mathcal{F} an ingredient of \mathcal{G} . We define $\text{Exp}_{\mathcal{F} \triangleleft \mathcal{G}}$ by:*

$$\text{Exp}_{\mathcal{F} \triangleleft \mathcal{G}} = \{\varepsilon \in \text{Exp}^c \mid \vdash \varepsilon : \mathcal{F} \triangleleft \mathcal{G}\}.$$

We define the set $\text{Exp}_{\mathcal{G}}$ of well-typed \mathcal{G} -expressions by $\text{Exp}_{\mathcal{G} \triangleleft \mathcal{G}}$.

In [2], it was proved that the set of \mathcal{G} -expressions for a given polynomial functor \mathcal{G} has a coalgebraic structure:

$$\delta_{\mathcal{G}} : \text{Exp}_{\mathcal{G}} \rightarrow \mathcal{G}(\text{Exp}_{\mathcal{G}})$$

More precisely, in [215], which we refer to for the complete definition of $\delta_{\mathcal{G}}$, the authors defined a function $\delta_{\mathcal{F} \triangleleft \mathcal{G}} : \text{Exp}_{\mathcal{F} \triangleleft \mathcal{G}} \rightarrow \mathcal{F}(\text{Exp}_{\mathcal{G}})$ and then set $\delta_{\mathcal{G}} = \delta_{\mathcal{G} \triangleleft \mathcal{G}}$.

The coalgebraic structure on the set of expressions enabled the proof of a Kleene like theorem.

Theorem 1 (Kleene’s theorem for polynomial coalgebras). *Let \mathcal{G} be a polynomial functor.*

1. *For any $\varepsilon \in \text{Exp}_{\mathcal{G}}$, there exists a finite \mathcal{G} -coalgebra (S, g) and $s \in S$ such that $\varepsilon \sim s$.*
2. *For every \mathcal{G} -coalgebra (S, g) and $s \in S$ there exists an expression $\varepsilon_s \in \text{Exp}_{\mathcal{G}}$ such that $\varepsilon_s \sim s$.*

In order to provide the reader we intuition over the notions presented above, we illustrate them with an example.

Example 1. Let us instantiate the definition of \mathcal{G} -expressions to the functors of streams $\mathcal{R} = \mathbf{B} \times \text{Id}$ (the ingredients of this functor are \mathbf{B} , Id and \mathcal{R} itself). Let X be a set of (recursion or) fixed-point variables. The set $\text{Exp}_{\mathcal{R}}$ of *stream expressions* is given by the set of closed and guarded expressions generated by the following BNF grammar. For $x \in X$:

$$\begin{aligned} \text{Exp}_{\mathcal{R}} \ni \varepsilon &::= \underline{\emptyset} \mid \varepsilon \oplus \varepsilon \mid \mu x. \varepsilon \mid x \mid l\langle \varepsilon_1 \rangle \mid r\langle \varepsilon \rangle \\ \varepsilon_1 &::= \underline{\emptyset} \mid b \mid \varepsilon_1 \oplus \varepsilon_1 \end{aligned} \tag{4}$$

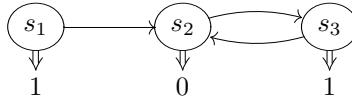
Intuitively, the expression $l\langle b \rangle$ is used to specify that the head of the stream is b , while $r\langle \varepsilon \rangle$ specifies a stream whose tail behaves as specified by ε . For the two element join-semilattice $\mathbf{B} = \{0, 1\}$ (with $\perp_{\mathbf{B}} = 0$), examples of well-typed expressions include $\underline{\emptyset}$, $l\langle 1 \rangle \oplus r\langle l\langle \underline{\emptyset} \rangle \rangle$ and $\mu x. r\langle x \rangle \oplus l\langle 1 \rangle$. The expressions $l[1]$, $l\langle 1 \rangle \oplus 1$ and $\mu x. 1$ are examples of non well-typed expressions for \mathcal{R} , because the functor \mathcal{R} does not involve \oplus , the subexpressions in the sum have different type, and recursion is not at the outermost level (1 has type $\mathbf{B} \triangleleft \mathcal{R}$), respectively.

By applying the definition in [2], the coalgebra structure on expressions $\delta_{\mathcal{R}}$ would be given by:

$$\begin{aligned}
 \delta_{\mathcal{R}} : \text{Exp}_{\mathcal{R}} &\rightarrow \mathbf{B} \times \text{Exp}_{\mathcal{R}} \\
 \delta_{\mathcal{R}}(\underline{\emptyset}) &= \langle 0, \underline{\emptyset} \rangle \\
 \delta_{\mathcal{R}}(\varepsilon_1 \oplus \varepsilon_2) &= \langle b_1 \vee b_2, \varepsilon'_1 \oplus \varepsilon'_2 \rangle \text{ where } \langle b_i, \varepsilon_i \rangle = \delta_{\mathcal{R}}(\varepsilon_i), \quad i = 1, 2 \\
 \delta_{\mathcal{R}}(\mu x. \varepsilon) &= \delta_{\mathcal{R}}(\varepsilon[\mu x. \varepsilon/x]) \\
 \delta_{\mathcal{R}}(l\langle \varepsilon_1 \rangle) &= \langle \delta_{\mathbf{B} \triangleleft \mathcal{R}}(\varepsilon_1), \underline{\emptyset} \rangle \\
 \delta_{\mathcal{R}}(r\langle \varepsilon \rangle) &= \langle \perp_{\mathbf{B}}, \varepsilon \rangle \\
 \delta_{\mathbf{B} \triangleleft \mathcal{R}}(\underline{\emptyset}) &= \perp_{\mathbf{B}} \\
 \delta_{\mathbf{B} \triangleleft \mathcal{R}}(b) &= b \\
 \delta_{\mathbf{B} \triangleleft \mathcal{R}}(\varepsilon_1 \oplus \varepsilon'_1) &= \delta_{\mathbf{B} \triangleleft \mathcal{R}}(\varepsilon_1) \vee \delta_{\mathbf{B} \triangleleft \mathcal{R}}(\varepsilon'_1)
 \end{aligned}$$

The proof of Kleene’s theorem provides algorithms to go from expressions to streams and vice-versa. We illustrate it by means of examples.

Consider the following stream:



We draw the stream with an automata-like flavor. The transitions indicate the tail of the stream represented by a state and the output value the head. In a more traditional notation, the above automata represents the infinite stream $(1, 0, 1, 0, 1, 0, 1, \dots)$.

To compute expressions $\varepsilon_1, \varepsilon_2$ and ε_3 equivalent to s_1, s_2 and s_3 we associate with each state s_i a variable x_i and we solve the following system of 3 equations in 3 variables:

$$\varepsilon_1 = \mu x_1. l\langle 1 \rangle \oplus r\langle x_2 \rangle \quad \varepsilon_2 = \mu x_2. l\langle 0 \rangle \oplus r\langle x_3 \rangle \quad \varepsilon_3 = \mu x_3. l\langle 1 \rangle \oplus r\langle x_2 \rangle$$

which yields the following closed expressions:

$$\varepsilon_1 = \mu x_1. l\langle 1 \rangle \oplus r\langle \varepsilon_2 \rangle \quad \varepsilon_2 = \mu x_2. l\langle 0 \rangle \oplus r\langle \varepsilon_3 \rangle \quad \varepsilon_3 = \mu x_3. l\langle 1 \rangle \oplus r\langle \mu x_2. l\langle 0 \rangle \oplus r\langle x_3 \rangle \rangle$$

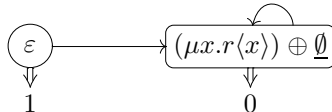
satisfying, by construction, $\varepsilon_1 \sim s_1, \varepsilon_2 \sim s_2$ and $\varepsilon_3 \sim s_3$.

For the converse construction, consider the expression $\varepsilon = (\mu x. r\langle x \rangle) \oplus l\langle 1 \rangle$. We construct an automaton by repeatedly applying the coalgebra structure on expressions $\delta_{\mathcal{R}}$, modulo ACI (associativity, commutativity and idempotency of \oplus) in order to guarantee finiteness.

Applying the definition of $\delta_{\mathcal{R}}$ above, we have:

$$\delta_{\mathcal{R}}(\varepsilon) = \langle 1, (\mu x. r\langle x \rangle) \oplus \underline{\emptyset} \rangle \text{ and } \delta_{\mathcal{R}}((\mu x. r\langle x \rangle) \oplus \underline{\emptyset}) = \langle 0, (\mu x. r\langle x \rangle) \oplus \underline{\emptyset} \rangle$$

which leads to the following stream (automaton):



Note that, throughout the paper, we will use streams as a basic example to illustrate the definitions. It should be remarked that the framework is general enough to include more complex examples, such as deterministic automata, automata on guarded strings or Mealy machines. The latter will be used as example in Section 4.1

3 An Algebraic View on the Coalgebra of Generalized Regular Expressions

We now have a (theoretical) framework which, given a functor \mathcal{G} , allows for the uniform derivation of 1) a language $\text{Exp}_{\mathcal{G}}$ for specifying behaviors of \mathcal{G} -systems, and 2) a coalgebraic structure on $\text{Exp}_{\mathcal{G}}$, which provides an operational semantics to the set of expressions. In the rest of the paper, we will extend and adapt the framework of the previous section in order to:

- enable the implementation of a tool which allows for the automatic derivation of 1) and 2) above
- enable automatic reasoning on equivalence of specifications; the reasoning will be performed by the coinductive prover CIRC [12], which is also the core of our target tool.

CIRC is based on algebraic specifications and, therefore, to reach our final goal we need two things:

- *algebraic specifications* that model both the language and the coalgebraic structure of expressions associated to polynomial functors to provide to CIRC
- a decision procedure, implemented in CIRC based on an *equational entailment relation*, in order to check for the bisimilarity of expressions.

We further give the basic notions the reader needs in order to get an easier understanding of the algebraic approach. An *algebraic specification* is a triple $\mathcal{E} = (S, \Sigma, E)$, where S is a set of *sorts*, Σ is a *many-sorted signature* and E is a set of *conditional equations* of the form $(\forall X) t = t' \text{ if } (\bigwedge_{i \in I} u_i = v_i)$, where t, t', u_i , and v_i ($i \in I$ – a set of indexes for the conditions) are Σ -terms with variables in X . We say that the *sort of the equation* is s whenever $t, t' \in \mathcal{T}_{\Sigma, s}(X)$. Here, $\mathcal{T}_{\Sigma, s}(X)$ denotes the set of terms of sort s of the Σ -algebra freely generated by X . If $I = \{\}$ then the equation is *unconditional* and may be written as $(\forall X) t = t'$.

Let \vdash be the *equational entailment (deduction) relation* defined as in [5]. We write $\mathcal{E} \vdash e$ whenever equation e is deducible from \mathcal{E} . We extend \mathcal{E} by adding the freezing operation $\boxed{-}: s \rightarrow \text{Frozen}$ for each sort $s \in \Sigma$, where **Frozen** is a fresh sort. By \boxed{t} we represent the *frozen* form of a Σ -term t , and by \boxed{c} a *frozen equation* of the shape $(\forall X) \boxed{t} = \boxed{t'}$ if c . The entailment relation \vdash is defined over frozen equations as in [12]. The need for the frozen operator will become clear in Example 2: without it the congruence rule could be applied freely leading to the derivation of untrue equations.

Fig. 1 briefly illustrates the parallel between the coalgebraic concepts presented in [15, 2] and their algebraic correspondents. In what follows, we will

coalgebraic	algebraic
$\vdash \varepsilon : \mathcal{F} \triangleleft \mathcal{G}$	$\mathcal{E}_{\mathcal{G}} \vdash \varepsilon : \mathcal{F} \triangleleft \mathcal{G} = true$
$\text{Exp}_{\mathcal{F} \triangleleft \mathcal{G}}$	$\{\varepsilon \in \mathcal{T}_{\Sigma, \text{Exp}} \mid \mathcal{E}_{\mathcal{G}} \vdash \varepsilon : \mathcal{F} \triangleleft \mathcal{G} = true\}$
$\text{Exp}_{\mathcal{G}}$	$\{\varepsilon \in \mathcal{T}_{\Sigma, \text{Exp}} \mid \mathcal{E}_{\mathcal{G}} \vdash \varepsilon : \mathcal{G} \triangleleft \mathcal{G} = true\}$
$\mathcal{F}(\text{Exp}_{\mathcal{G}})$	$\{\sigma \in \mathcal{T}_{\Sigma, \text{ExpStruct}} \mid \mathcal{E}_{\mathcal{G}} \vdash \sigma : \mathcal{F}(\text{Exp}_{\mathcal{G}}) = true\}$
$\delta_{\mathcal{F} \triangleleft \mathcal{G}} : \text{Exp}_{\mathcal{F} \triangleleft \mathcal{G}} \rightarrow \mathcal{F}(\text{Exp}_{\mathcal{G}})$	$\delta_{-}(_): \text{Ingredient Exp} \rightarrow \text{ExpStruct}$
$\langle \sigma, \sigma' \rangle \in \overline{\mathcal{F}}(cl(\mathcal{R}_{id}))$	$\mathcal{E}_{\mathcal{G}} \vdash \sigma : \mathcal{F}(\text{Exp}_{\mathcal{G}}) = true, \mathcal{E}_{\mathcal{G}} \vdash \sigma' : \mathcal{F}(\text{Exp}_{\mathcal{G}}) = true$ $\mathcal{E}_{\mathcal{G}} \cup \boxed{\mathcal{R}} \vdash_{PF} \boxed{\sigma} = \boxed{\sigma'}$ (i)
$cl(\mathcal{R}_{id})$ is a bisimulation	$\mathcal{E}_{\mathcal{G}} \cup \boxed{\mathcal{R}} \vdash_{PF} \boxed{\delta_{\mathcal{G} \triangleleft \mathcal{G}}(\mathcal{R})}$ (ii)

Fig. 1. Polynomial functors - coalgebraic vs. algebraic approach

provide some explanations on the algebraic side, in order to model what we presented coalgebraically in the previous section, analyzing the components of Fig. 1.

The algebraic specification of a polynomial functor. For the provided functor \mathcal{G} , the specification $\mathcal{E}_{\mathcal{G}} = (S, \Sigma, E)$ is incrementally built according to the items common to all regular expressions, extended with the items specific to \mathcal{G} (e.g., the semilattices, the exponentiation alphabets). As an initial step in the construction of $\mathcal{E}_{\mathcal{G}}$, we use the general rule for translating definitions based on Backus-Naur grammars into algebraic specifications. Each syntactical category and vocabulary is considered as a sort, and each production is considered as a constructor operation or a subsort relation. For instance, according to the grammar of generalized regular expressions in Definition 3, we have: a sort **Exp** representing expressions ε , **FixpVar** the sort for the vocabulary of the fixed-point variables, **Alph** the sort for the elements of the alphabets, and **Slit** the sort for the elements of the semilattices. Moreover, we consider constructor operations for all the productions. For example, the production $\varepsilon ::= \varepsilon \oplus \varepsilon$ is represented by an operation $_ \oplus _ : \text{Exp Exp} \rightarrow \text{Exp}$. Using a similar mechanism, we specify:

- structured expressions σ , the counterpart of $\mathcal{F}(\text{Exp}_{\mathcal{G}})$, defined by
$$\sigma ::= \varepsilon \mid \langle \sigma, \sigma \rangle \mid k_1(\sigma) \mid k_2(\sigma) \mid \perp \mid \top \mid \lambda x.(a, \mathcal{F} \triangleleft \mathcal{G}, \sigma)$$
we denote the sort of this kind of expressions by **ExpStruct** (the construction $\lambda x.(a, \mathcal{F} \triangleleft \mathcal{G}, \sigma)$ has as coalgebraic correspondent a function $f \in \mathcal{F}^A(\text{Exp}_{\mathcal{G}})$)
- polynomial functors defined by grammar (1); the associated sort is **Functor**
- functor ingredients given in Definition 2; the corresponding sort is **Ingredient**

The set $\text{Exp}_{\mathcal{F} \triangleleft \mathcal{G}}$ of expressions of type $\mathcal{F} \triangleleft \mathcal{G}$ is algebraically represented by the set of Σ -terms ε of sort **Exp**, such that $\mathcal{E}_{\mathcal{G}} \vdash \varepsilon : \mathcal{F} \triangleleft \mathcal{G} = true$. The type-checking relation in Definition 4 is given by an operation $_ : _ : \text{Exp Ingredient} \rightarrow \text{Bool}$ and an equation for each inference rule defining this relation. For example

$$\frac{\vdash \varepsilon_1 : \mathcal{F} \triangleleft \mathcal{G} \quad \vdash \varepsilon_2 : \mathcal{F} \triangleleft \mathcal{G}}{\vdash \varepsilon_1 \oplus \varepsilon_2 : \mathcal{F} \triangleleft \mathcal{G}}$$

is represented by the equation $\varepsilon_1 \oplus \varepsilon_2 : \mathcal{F} \triangleleft \mathcal{G} = \varepsilon_1 : \mathcal{F} \triangleleft \mathcal{G} \wedge \varepsilon_2 : \mathcal{F} \triangleleft \mathcal{G}$. For the sake of notation, algebraically we write $\varepsilon : \mathcal{F} \triangleleft \mathcal{G}$ to represent expressions of type $\mathcal{F} \triangleleft \mathcal{G}$.

The structured expressions $\sigma \in \mathcal{F}(\text{Exp}_{\mathcal{G}})$ are given by the set of Σ -terms of sort ExpStruct , such that $\mathcal{E}_{\mathcal{G}} \vdash \sigma : \mathcal{F}(\text{Exp}_{\mathcal{G}}) = \text{true}$ (here $:$ is the extension of the type-checking operator to structured expressions). Algebraically, we write $\sigma : \mathcal{F}(\text{Exp}_{\mathcal{G}})$ to denote that σ is an element of $\mathcal{F}(\text{Exp}_{\mathcal{G}})$.

The function $\delta_{\mathcal{G}}$, which provides the coalgebraic structure of \mathcal{G} -expressions, has the algebraic correspondent $\delta \in \Sigma$, a function parameterized with the functor ingredients.

Recall from Section 2 that a relation $\mathcal{R} \subseteq \text{Exp}_{\mathcal{G} \triangleleft \mathcal{G}} \times \text{Exp}_{\mathcal{G} \triangleleft \mathcal{G}}$ is a bisimulation if and only if $(s, t) \in \mathcal{R} \Rightarrow \langle \delta_{\mathcal{G} \triangleleft \mathcal{G}}(s), \delta_{\mathcal{G} \triangleleft \mathcal{G}}(t) \rangle \in \overline{\mathcal{G}}(\mathcal{R})$. In order to enable the algebraic framework to decide bisimilarity of \mathcal{G} -expressions, we define a new entailment relation for polynomial functors \vdash_{PF} (the definitions of $\overline{\mathcal{G}}$ and \vdash_{PF} are closely related).

Definition 6. *The entailment relation \vdash_{PF} is the extension of \vdash with the following inference rules, which allow a restricted contextual reasoning over the frozen equations of structured expressions:*

$$\frac{\mathcal{E}_{\mathcal{G}} \vdash_{PF} \boxed{\sigma_1} = \boxed{\sigma'_1} \quad \mathcal{E}_{\mathcal{G}} \vdash_{PF} \boxed{\sigma_2} = \boxed{\sigma'_2}}{\mathcal{E}_{\mathcal{G}} \vdash_{PF} \boxed{\langle \sigma_1, \sigma_2 \rangle} = \boxed{\langle \sigma'_1, \sigma'_2 \rangle}} \quad (5)$$

$$\frac{\mathcal{E}_{\mathcal{G}} \vdash_{PF} \boxed{\sigma} = \boxed{\sigma'}}{\mathcal{E}_{\mathcal{G}} \vdash_{PF} \boxed{k_i(\sigma)} = \boxed{k_i(\sigma')}} \quad (i = \overline{1, 2}) \quad (6)$$

$$\frac{\mathcal{E}_{\mathcal{G}} \vdash_{PF} \boxed{f(a)} = \boxed{g(a)}, \text{ for all } a \in A}{\mathcal{E}_{\mathcal{G}} \vdash_{PF} \boxed{f} = \boxed{g}} \quad (7)$$

Let \mathcal{G} be a polynomial functor, and \mathcal{R} a binary relation on the set of \mathcal{G} -expressions. We will make use of the conventions:

- $\mathcal{R}_{id} = \mathcal{R} \cup \{(\varepsilon, \varepsilon) \mid \mathcal{E}_{\mathcal{G}} \vdash \varepsilon : \mathcal{G} \triangleleft \mathcal{G} = \text{true}\}$
- $cl(\mathcal{R})$ is the closure of \mathcal{R} under transitivity, symmetry and reflexivity
- $\boxed{\mathcal{R}} = \bigcup_{e \in \mathcal{R}} \{\boxed{e}\}$ (application of the freezing operator to all elements of \mathcal{R})
- $\mathcal{E}_{\mathcal{G}} \cup \boxed{\mathcal{R}}$ is a shorthand for $(S, \Sigma, E \cup \{\boxed{e} = \boxed{e'} \mid (\varepsilon, \varepsilon') \in \mathcal{R}\})$
- $\delta_{\mathcal{G} \triangleleft \mathcal{G}}(\varepsilon = \varepsilon')$ denotes the equation $\delta_{\mathcal{G} \triangleleft \mathcal{G}}(\varepsilon) = \delta_{\mathcal{G} \triangleleft \mathcal{G}}(\varepsilon')$
- $\langle \sigma, \sigma' \rangle \in \overline{\mathcal{G}}(\mathcal{R})$ is a shorthand for: (σ, σ') is an element of the set S , where $\mathcal{E}_{\mathcal{G}} \vdash \overline{\mathcal{G}}(\mathcal{R}) = S$ (here, $\overline{\mathcal{G}}(\mathcal{R}) \subseteq \mathcal{T}_{\Sigma, \text{ExpStruct}} \times \mathcal{T}_{\Sigma, \text{ExpStruct}}$)

The following theorem and corollary correspond to the equivalences (i), and respectively (ii), in Fig. 1. Theorem 2 formalizes the connection between the inductive definition of $\overline{\mathcal{G}}$ (on the coalgebraic side) and \vdash_{PF} (on the algebraic side), hence enabling the definition of bisimulations in algebraic terms, in Corollary 1.

Theorem 2. Consider a polynomial functor \mathcal{G} and \mathcal{F} an ingredient of \mathcal{G} . If \mathcal{R} is a binary relation on the set of \mathcal{G} -expressions, and $\sigma, \sigma' : \mathcal{F}(\text{Exp}_{\mathcal{G}})$ then $\langle \sigma, \sigma' \rangle \in \overline{\mathcal{F}}(cl(\mathcal{R}_{id}))$ iff $\mathcal{E}_{\mathcal{G}} \cup \boxed{\mathcal{R}} \vdash_{PF} \boxed{\sigma} = \boxed{\sigma'}$.

Proof. The proof is by induction on the structure of \mathcal{F} . Take, for example the direct implication “ \Rightarrow ”. The base case $\mathcal{F} = \mathbf{B}$ holds by the reflexivity of \vdash_{PF} . The case $\mathcal{F} = \text{Id}$ follows immediately according to an auxiliary result stating that if $(\varepsilon, \varepsilon') \in cl(\mathcal{R}_{id})$ then $\mathcal{E}_{\mathcal{G}} \cup \boxed{\mathcal{R}} \vdash_{PF} \boxed{\varepsilon} = \boxed{\varepsilon'}$. Inductive steps hold by the rules (5), (6) and (7), defining \vdash_{PF} . A similar reasoning is used for proving “ \Leftarrow ”. \square

Corollary 1. Let \mathcal{G} be a polynomial functor. If \mathcal{R} is a binary relation on the set of \mathcal{G} -expressions, then $cl(\mathcal{R}_{id})$ is a bisimulation iff $\mathcal{E}_{\mathcal{G}} \cup \boxed{\mathcal{R}} \vdash_{PF} \boxed{\delta_{\mathcal{G} \triangleleft \mathcal{G}}(\mathcal{R})}$.

Proof. The result follows immediately according to the equivalences:

$$cl(\mathcal{R}_{id}) \text{ is a bisimulation} \Leftrightarrow_{\text{(Definition 1)}} (\forall (\varepsilon, \varepsilon') \in cl(\mathcal{R}_{id})). \langle \delta_{\mathcal{G} \triangleleft \mathcal{G}}(\varepsilon), \delta_{\mathcal{G} \triangleleft \mathcal{G}}(\varepsilon') \rangle \in \overline{\mathcal{G}}(cl(\mathcal{R}_{id})) \Leftrightarrow_{\text{(Theorem 2)}} \mathcal{E}_{\mathcal{G}} \cup \boxed{\mathcal{R}} \vdash_{PF} \boxed{\delta_{\mathcal{G} \triangleleft \mathcal{G}}(cl(\mathcal{R}_{id}))} \Leftrightarrow_{\text{(def. } cl(\mathcal{R}_{id}), \vdash_{PF})} \mathcal{E}_{\mathcal{G}} \cup \boxed{\mathcal{R}} \vdash_{PF} \boxed{\delta_{\mathcal{G} \triangleleft \mathcal{G}}(\mathcal{R})}. \quad \square$$

4 A Decision Procedure for Bisimilarity

In this section we describe how the coinductive theorem prover CIRC [11] can be used to implement a decision procedure for the bisimilarity of generalized regular expressions.

CIRC can be seen as an extension of Maude with behavioral features and its implementation is derived from that of Full-Maude. In order to use the prover, one needs to provide a specification (a CIRC theory) and a set of goals. A CIRC theory $\mathcal{B} = (S, (\Sigma, \Delta), (E, \mathcal{I}))$ consists of an algebraic specification (S, Σ, E) , a set Δ of derivatives (= Σ -contexts), and a set \mathcal{I} of equational interpolants, which are expressions of the form $e \Rightarrow \{e_i \mid i \in I\}$ where e and e_i are equations (for more information on equational interpolants see [6]). A derivative $\delta \in \Delta$ is a Σ -term containing a special variable $*:s$, where s is the sort of the variable $*$. If e is an equation $t = t'$ with t and t' of sort s , then $\delta[e]$ is $\delta[t/*:s] = \delta[t'/*:s]$. Let $\Delta[e]$ denote the set $\{\delta[e] \mid \delta \in \Delta \text{ appropriate for } e\}$.

CIRC implements the coinductive proof system given in [12] using a set of reduction rules of the form $(\mathcal{B}, \mathcal{F}, \mathcal{G}) \Rightarrow (\mathcal{B}, \mathcal{F}', \mathcal{G}')$, where \mathcal{B} represents a specification, \mathcal{F} is the coinductive hypothesis (a set of frozen equations) and \mathcal{G} is the current set of goals. The freezing operator is defined as described in Section 3. Here is a brief description of these rules:

[Done]: $(\mathcal{B}, \mathcal{F}, \{\}) \Rightarrow \cdot$

Whenever the set of goals is empty, the system terminates with success.

[Reduce]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\square\}) \Rightarrow (\mathcal{B}, \mathcal{F}, \mathcal{G})$ if $\mathcal{B} \cup \mathcal{F} \vdash \square$

If the current goal is a \vdash -consequence of $\mathcal{B} \cup \mathcal{F}$ then \square is removed from the set of goals.

[Derive]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{e}\}) \Rightarrow (\mathcal{B}, \mathcal{F} \cup \{\boxed{e}\}, \mathcal{G} \cup \{\boxed{\Delta[e]}\})$ if $\mathcal{B} \cup \mathcal{F} \not\models \boxed{e}$

When the current goal e has the same sort with the special variable $*$, and it is not a \vdash -consequence, it is added to the specification and its derivatives to the set of goals. In order to simplify the notation, we write $\delta(e)$ for $\delta(\varepsilon) = \delta(\varepsilon')$, whenever e is of shape $\varepsilon = \varepsilon'$.

[Simplify]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{\theta(e)}\}) \Rightarrow (\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{\theta(e_i)} \mid i \in I\})$
 if $e \Rightarrow \{e_i \mid i \in I\}$ is a simplification rule from the specification and $\theta: X \rightarrow \mathcal{T}_Y(Y)$ is a substitution.

[Fail]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{e}\}) \Rightarrow \text{failure}$ if $\mathcal{B} \cup \mathcal{F} \not\models \boxed{e} \wedge e:\text{Bool}$

This rule stops the reduction process with failure whenever the current goal e is of type **Bool** and the corresponding normal forms are different.

It is worth noting that there is a strong connection between a CIRC proof and the construction of a bisimulation relation. We emphasize this fact and the importance of the freezing operator with a simple example.

Example 2. Consider the case of infinite streams. The set \mathbf{B}^ω of infinite streams over a set \mathbf{B} is the final coalgebra of the functor $\mathcal{R} = \mathbf{B} \times \text{Id}$, with a coalgebra structure given by hd and tl , the functions that return the head and the tail of the stream, respectively. Our purpose is to prove that $0^\infty = (00)^\infty$. Let z and zz represent the stream on the left hand side and, respectively, on the right hand side. These streams are defined by the equations: $hd(z) = 0, tl(z) = z, hd(zz) = 0, tl(zz) = 0:zz$. In Fig. 2 we present the correlation between the CIRC proof and the construction of the bisimulation relation. Note how CIRC collects the elements of the bisimulation as frozen hypothesis.

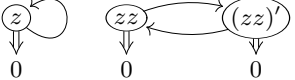
CIRC proof	Bisimulation construction
(add goal $z = zz$.)	
$(\mathcal{B}, \{\}, \{\boxed{z} = \boxed{zz}\})$	$\mathcal{F} = \{\}; z \sim zz ?$
$\xrightarrow{[\text{Derive}]}$ $(\mathcal{B}, \{\boxed{z} = \boxed{zz}\}, \left\{ \begin{array}{l} \boxed{hd(z)} = \boxed{hd(zz)} \\ \boxed{tl(z)} = \boxed{tl(zz)} \end{array} \right\})$	$\mathcal{F} = \{(z, zz)\}; z \xrightarrow{0} z, zz \xrightarrow{0} (zz)'$
$\xrightarrow{[\text{Reduce}]}$ $(\mathcal{B}, \{\boxed{z} = \boxed{zz}\}, \{\boxed{z} = \boxed{0:zz}\})$	$\mathcal{F} = \{(z, zz)\}; z \sim (zz)' ?$
$\xrightarrow{[\text{Derive}]}$ $(\mathcal{B}, \left\{ \begin{array}{l} \boxed{z} = \boxed{zz} \\ \boxed{z} = \boxed{0:zz} \end{array} \right\}, \left\{ \begin{array}{l} \boxed{hd(z)} = \boxed{hd(0:zz)} \\ \boxed{tl(z)} = \boxed{tl(0:zz)} \end{array} \right\})$	$\mathcal{F} = \{(z, zz), (z, (zz)')\}; z \xrightarrow{0} z, (zz)' \xrightarrow{0} zz$
$\xrightarrow{[\text{Reduce}]}$ $(\mathcal{B}, \left\{ \begin{array}{l} \boxed{z} = \boxed{zz} \\ \boxed{z} = \boxed{0:zz} \end{array} \right\}, \{\})$	$\mathcal{F} = \{(z, zz), (z, (zz)')\} \checkmark$

Fig. 2. Parallel between a CIRC proof and the bisimulation construction

Let us analyze what happens if the freezing operator $\boxed{}$ would not be used. Suppose the circular coinduction algorithm would add the equation $z = zz$ in its unfrozen form to the hypothesis. After applying the derivatives we obtain the goals $hd(z) = hd(zz)$, $tl(z) = tl(zz)$. At this point, the prover could use the freshly added equation, and according to the congruence rule, both goals would be proven directly, though we would still be in the process of showing that the hypothesis holds. By following a similar reasoning, we could then also prove that $0^\infty = 1^\infty$! In order to avoid these situations, the hypotheses are frozen (*i.e.*, their sort is changed from **Stream** to **Frozen**) and this stops the application of the congruence rule, forcing the application of the derivatives according to their definition in the specification. Therefore, the use of the freezing operator is vital for the soundness of circular coinduction.

Next, we focus on using CIRC for automatically reasoning on the equivalence of \mathcal{G} -expressions. As we will show, the implementation of both the algebraic specifications associated to polynomial functors and the equational entailment relation described in Section 3, is immediate. Given a polynomial functor \mathcal{G} , we define a CIRC theory $\mathcal{B}_\mathcal{G} = (S, (\Sigma, \Delta), (E, \mathcal{I}))$ as follows:

- (S, Σ, E) is $\mathcal{E}_\mathcal{G}$
- $\Delta = \{\delta_{\mathcal{G} \triangleleft \mathcal{G}}(*:\text{Exp})\}$
- \mathcal{I} consists of the following equational interpolants:

$$\langle \sigma_1, \sigma_2 \rangle = \langle \sigma'_1, \sigma'_2 \rangle \Rightarrow \{ \sigma_1 = \sigma'_1, \sigma_2 = \sigma'_2 \} \tag{8}$$

$$\{ k_i(\sigma) = k_i(\sigma') \} \Rightarrow \{ \sigma = \sigma' \} \tag{9}$$

$$\{ f = g \} \Rightarrow \{ f(a) = g(a) \mid a \in A \} \tag{10}$$

The interpolants (8), (9) and (10) in \mathcal{I} extend the entailment relation \vdash from the system above to \vdash_{PF} (see Definition 6) as follows:

$$\frac{E \vdash e}{E \vdash_{PF} e} \qquad \frac{E \vdash_{PF} \{e_i \mid i \in I\}}{E \vdash_{PF} e} \text{ if } e \Rightarrow \{e_i \mid i \in I\} \text{ in } \mathcal{I}$$

Theorem 3 (Soundness). *Let \mathcal{G} be a polynomial functor, and \mathcal{G} a binary relation on the set of \mathcal{G} -expressions. If $(\mathcal{B}_\mathcal{G}, \mathcal{F}_0 = \{\}, \mathcal{G}_0 = \boxed{\mathcal{G}}) \xrightarrow{*} (\mathcal{B}_\mathcal{G}, \mathcal{F}_n, \mathcal{G}_n = \{\})$ using [Reduce], [Derive] and [Simplify], then $\mathcal{G} \subseteq \sim_\mathcal{G}$.*

Proof. The idea of the proof is to identify a bisimulation relation $\tilde{\mathcal{F}}$ s.t. $\mathcal{G} \subseteq \tilde{\mathcal{F}}$. On a closer look, based on the reduction rules implemented in CIRC, it is quite easy to see that the initial set of goals \mathcal{G} is a \vdash_{PF} -consequence of $\mathcal{B}_\mathcal{G} \cup \boxed{\mathcal{F}}$, where \mathcal{F} is the set of hypothesis (or derived goals) collected during a proof session. In other words, $\mathcal{G} \subseteq cl(\mathcal{F}_{id})$. So, if we anticipate a bit, we should show that $\tilde{\mathcal{F}} = cl(\mathcal{F}_{id})$ is a bisimulation, *i.e.*, according to Corollary 1, $\mathcal{B}_\mathcal{G} \cup \boxed{\mathcal{F}} \vdash_{PF} \boxed{\delta_{\mathcal{G} \triangleleft \mathcal{G}}(\mathcal{F})}$. This is achieved by proving that $\mathcal{B}_\mathcal{G} \cup \boxed{\mathcal{F}} \vdash_{PF} \mathcal{G}_i (i = \overline{0..n})$ (note that $\boxed{\delta_{\mathcal{G} \triangleleft \mathcal{G}}(\mathcal{F})} \subseteq \bigcup_{i=\overline{0..n}} \mathcal{G}_i$, according to [Derive]). The demonstration is by induction on j , where $n - j$ is the current proof step, and by case analysis on the CIRC reduction rules applied at each step. \square

Remark 1. The soundness of the proof system we describe in this paper does not follow directly from Theorem 3 in [12]. This is due to the fact that we do not have

an experiment-based definition of bisimilarity. So, even though the mechanism we use for proving $\mathcal{B}_{\mathcal{G}} \cup \boxed{\mathcal{F}} \vdash_{PF} \boxed{\delta_{\mathcal{G} \triangleleft \mathcal{G}}(\mathcal{F})}$ is similar to the one described in [12], the current soundness proof is conceived in terms of bisimulations (and not experiments).

Remark 2. The entailment relation \vdash_{PF} CIRC uses for checking for the equivalence of generalized regular expressions is an instantiation of the parametric entailment relation \vdash from the proof system in [12]. This approach extends CIRC to automatically reason on a large class of systems that can be modeled as coalgebras of polynomial functors.

As already stated, our final purpose is to use CIRC as a decision procedure for the bisimilarity of generalized regular expressions. That is, whenever provided a set of expressions, the prover stops with an yes/no answer w.r.t. their equivalence. In this context, an important aspect is that the sub-coalgebra generated by an expression $\varepsilon \in \text{Exp}_{\mathcal{G}}$ by repeatedly applying $\delta_{\mathcal{G} \triangleleft \mathcal{G}}$ is, in general, infinite. Take for example the polynomial functor $\mathcal{G} = \mathbf{B} \times \text{Id}$ associated to infinite streams, and consider the property $\mu x. \underline{0} \oplus r\langle x \rangle = \mu x. r\langle x \rangle$. In order to prove this, CIRC builds an infinite proof sequence by repeatedly applying $\delta_{\mathcal{G} \triangleleft \mathcal{G}}$ as follows:

$$\begin{aligned} \delta_{\mathcal{G} \triangleleft \mathcal{G}}(\mu x. \underline{0} \oplus r\langle x \rangle) &= \delta_{\mathcal{G} \triangleleft \mathcal{G}}(\mu x. r\langle x \rangle) \\ &\downarrow \\ \langle 0, \underline{0} \oplus (\mu x. \underline{0} \oplus r\langle x \rangle) \rangle &= \langle 0, \mu x. r\langle x \rangle \rangle \\ \delta_{\mathcal{G} \triangleleft \mathcal{G}}(\underline{0} \oplus (\mu x. \underline{0} \oplus r\langle x \rangle)) &= \delta_{\mathcal{G} \triangleleft \mathcal{G}}(\mu x. r\langle x \rangle) \\ &\downarrow \\ \langle 0, \underline{0} \oplus \underline{0} \oplus (\mu x. \underline{0} \oplus r\langle x \rangle) \rangle &= \langle 0, \mu x. r\langle x \rangle \rangle [\dots] \end{aligned}$$

In this case, the prover would never stop. It is shown in [215] that the axioms for associativity, commutativity and idempotency (ACI) guarantee finiteness of the generated sub-coalgebra (note that these axioms have also been proven sound w.r.t. bisimulation). ACI properties can easily be specified in CIRC as the prover is an extension of Maude, which has a powerful matching modulo ACUI capability. The idempotency is given by the equation $\varepsilon \oplus \varepsilon = \varepsilon$, and the commutativity and associativity are specified as attributes of \oplus .

Theorem 4. *Let \mathcal{G} be a set of proof obligations over generalized regular expressions. CIRC can be used as a decision procedure for the equivalences in \mathcal{G} , that is, it can assert whenever a goal $(\varepsilon_1, \varepsilon_2) \in \mathcal{G}$ is a true or false equality.*

Proof. The result is a consequence of the fact that by implementing the ACI axioms in CIRC, the set of new goals obtained by repeatedly applying the derivative δ is finite. In these circumstances, whenever CIRC stops according to the reduction rule [Done], the initial proof obligations are bisimilar. On the other hand, whenever it terminates with [Fail], the goals are not bisimilar. \square

4.1 A CIRC-Based Tool

We have implemented a tool that, when provided with a functor \mathcal{G} , automatically generates a specification for CIRC which can then be used in order to

automatically check whether two \mathcal{G} -expressions are bisimilar. The tool is implemented as a metalanguage application in Maude. It can be downloaded from <http://circidei.info.uaic.ro/functorizer/functorizer.maude>.

Let us now show another example: Mealy machines, which are coalgebras for the functor $(\mathbf{B} \times \text{Id})^A$. In what follows we show how CIRC can be used in conjunction with our tool in order to act as a decision procedure when checking for the equivalence of two expressions.

Formally, a Mealy machine is a pair (S, α) consisting of a set S of states and a transition function $\alpha: S \rightarrow (\mathbf{B} \times S)^A$, which for each state $s \in S$ and input $a \in A$ associates an output value b and a next state s' . Typically, we write $\alpha(s)(a) = \langle b, s' \rangle \Leftrightarrow (s \xrightarrow{a|b} s')$. As an example, consider the Mealy machine depicted in Fig. 3, where all the states are bisimilar.

We first show how to check for the equivalence of two expressions characterizing the states s_1 and s_2 from the Mealy machine in Fig. 3. These expressions, which could be computed, using the algorithm in Kleene's theorem, are $\varepsilon_1 = a(r\langle \mu x.a(r\langle x \rangle) \oplus b(\emptyset) \rangle) \oplus b(r\langle \mu y.a(r\langle y \rangle) \oplus b(r\langle y \rangle) \rangle)$ and $\varepsilon_2 = \mu x.a(r\langle x \rangle) \oplus b(r\langle x \rangle)$, respectively.

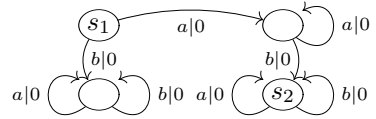


Fig. 3. Mealy machine: $s_1 \sim s_2$

In order to check for the bisimilarity of ε_1 and ε_2 we load the tool and define the semilattice $\mathbf{B} = \{0\}$ and the alphabet $A = \{a, b\}$:

```
(jslt B is 0 bottom 0 . 0 v 0 = 0 . endjslt)
(alph A is a b endalph)
```

We provide the functor \mathcal{G} using the command `(functor (B x Id)^A .)`. The command `(set goal)` specifies the goal we want to prove:

```
(set goal a(r< μ X:FixpVar . a(r< X:FixpVar >) (+) b(∅)>) (+)
      b(r< μ Y:FixpVar . a(r< Y:FixpVar >) (+) b(r< Y:FixpVar >) >) =
      μ X:FixpVar . a(r< X:FixpVar >) (+) b(r< X:FixpVar >) .)
```

In order to generate the CIRC specification we use the command `(generate coalgebra .)`. Next we need to load CIRC along with the resulting specification and start the proving engine using the command `(coinduction .)`.

As already shown, behind the scenes, CIRC builds a bisimulation relation that includes the initial goal. The proof succeeds and the output consists of (a subset of) this bisimulation:

```
Proof succeeded.
Number of derived goals: 3
Number of proving steps performed: 82
[...]
Proved properties:
[...]
a(r< μ X . a(r< X >) (+) b(∅) >) (+)
b(r< μ Y . a(r< Y >) (+) b(r< Y >) >) =
μ X . a(r< X >) (+) b(r< X >)
```

As previously mentioned, CIRC is also able to detect when two expressions are not equivalent. Take, for instance, the expressions $\mu x.a(r\langle a(l\langle 1 \rangle) \oplus x \rangle)$ and $a(r\langle a(l\langle 1 \rangle) \rangle) \oplus \mu x.a(r\langle x \rangle)$, characterizing the states s_1 and s_3 from the Mealy machines in Fig. 4. After following some steps similar to the ones previously enumerated, the proof fails and the output message is `Visible goal [...] failed during coinduction.`

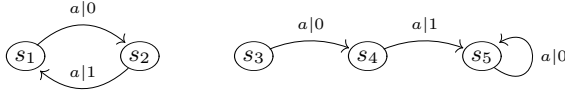


Fig. 4. Mealy machines: $s_1 \not\sim s_3$

5 Conclusions and Future Work

One of the major contributions of this paper is that we exploited an encoding of coalgebra into algebra, and provided a decision procedure for the bisimilarity of generalized regular expressions. In order to enable the implementation of the decision procedure, we formalized the equivalence between the coalgebraic concepts associated to polynomial coalgebras [21] and their algebraic correspondents. This led to the definition of algebraic specifications ($\mathcal{E}_\mathcal{G}$) that model both the language and the coalgebraic structure of expressions. Moreover, we defined an equational deduction relation (\vdash_{PF}), used on the algebraic side for reasoning on the bisimilarity of expressions.

The most important result of the parallel between the coalgebraic and algebraic approaches is given in Corollary 1, which formalizes the definition of the bisimulation relations, in algebraic terms. Actually, this result is the key for proving the soundness of the decision procedure implemented in the automated prover CIRC [11]. As a coinductive prover, CIRC builds a relation \mathcal{F} closed under the application of $\delta_\mathcal{G}$ with respect to $\vdash_{PF} (\mathcal{E}_\mathcal{G} \cup \boxed{\mathcal{F}} \vdash_{PF} \boxed{\delta_\mathcal{G}(\mathcal{F})})$, hence automatically computing a bisimulation the initial proof obligations belong to.

The approach we present in this paper enables CIRC to perform a reasoning based on bisimulations (instead of experiments [12]). This way, the prover is extended to checking for the bisimilarity in a large class of systems that can be modeled as \mathcal{G} -coalgebras. Note that the constructions above are all automated – the (non-trivial) CIRC algebraic specification describing $\mathcal{E}_\mathcal{G}$, together with the interpolants implementing \vdash_{PF} are generated with the Maude tool presented in Section 4.1.

As future work, we intend to extend our proof system to Kripke polynomial coalgebras and to exploit more of the axioms in [1] with the purpose of improving the prover’s time performance (our experience so far shows that by adding the axiom for the distribution of the $\underline{\emptyset}$ expression through the constructors, the prover works significantly faster).

Acknowledgments. The authors are grateful for useful comments from Filippo Bonchi and the anonymous reviewers. The work of Georgiana Caltais and Eugenio Goriac has been partially supported by the PNII grant CNCISIS IDEI 393 and the project ‘Meta-theory of Algebraic Process Theories’ (nr. 100014021) of the Icelandic Research Fund. The work of Dorel Lucanu has been partially supported by the PNII grant CNCISIS IDEI 393.

References

1. Bonsangue, M.M., Rutten, J.J.M.M., Silva, A.: An algebra for Kripke polynomial coalgebras. In: LICS, pp. 49–58. IEEE Computer Society, Los Alamitos (2009)
2. Bonsangue, M., Rutten, J., Silva, A.: A Kleene theorem for polynomial coalgebras. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 122–136. Springer, Heidelberg (2009)
3. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
4. Goguen, J., Lin, K., Rosu, G.: Circular coinductive rewriting. In: ASE 2000: Proceedings of the 15th IEEE International Conference on Automated Software Engineering, Washington, DC, USA, 2000, pp. 123–132. IEEE Computer Society, Los Alamitos (2000)
5. Goguen, J.A.: Order-sorted algebra i: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science* 105, 217–273 (1992)
6. Goriac, E.-I., Lucanu, D., Roşu, G.: Automating Coinduction with Case Analysis. Technical Report TR 10-05, “Al.I.Cuza” University of Iaşi, Faculty of Computer Science (2010), <http://www.infoiasi.ro/~tr/tr.pl.cgi>
7. Jacobs, B.: Introduction to coalgebra. towards mathematics of states and observations (2005)
8. Kleene, S.: Representation of events in nerve nets and finite automata. *Automata Studies*, 3–42 (1956)
9. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. In: LICS, pp. 214–225. IEEE Computer Society, Los Alamitos (1991)
10. Kozen, D.: Myhill-nerode relations on automatic systems and the completeness of Kleene algebra. In: Ferreira, A., Reichel, H. (eds.) STACS 2001. LNCS, vol. 2010, pp. 27–38. Springer, Heidelberg (2001)
11. Lucanu, D., Goriac, E.-I., Caltais, G., Roşu, G.: CIRC: A behavioral verification tool based on circular coinduction. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) CALCO 2009. LNCS, vol. 5728, pp. 433–442. Springer, Heidelberg (2009)
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. Rutten, J.J.M.M.: Universal coalgebra: a theory of systems. *Theor. Comput. Sci.* 249(1), 3–80 (2000)
14. Salomaa, A.: Two complete axiom systems for the algebra of regular events. *J. ACM* 13(1), 158–169 (1966)
15. Silva, A., Bonsangue, M.M., Rutten, J.J.M.M.: Non-deterministic kleene coalgebras. *Logical Methods in Computer Science* 6(3) (2010)

Normalization of Linear Horn Clauses

Thomas Martin Gawlitza¹, Helmut Seidl², and Kumar Neeraj Verma²

¹CNRS/VERIMAG

Thomas.Gawlitza@imag.fr*

²Institut für Informatik, TU München, Germany

{seidl, verma}@in.tum.de

Abstract. Nielson et al. [12] exhibit a rich class of Horn clauses which they call \mathcal{H}_1 . Least models of finite sets of \mathcal{H}_1 Horn clauses are regular tree languages. Nielson et al. [12] describe a normalization procedure for computing least models of finite sets of \mathcal{H}_1 Horn clauses in the form of tree automata. In the present paper, we simplify and extend this normalization procedure to a semi-procedure that deals with finite sets of *linear* Horn clauses. The extended semi-procedure does not terminate in general but does so on useful subclasses of finite sets of linear Horn clauses. The extension in particular coincides with the normalization procedure of Nielson et al. [12] for sets of \mathcal{H}_1 Horn clauses. In order to demonstrate the usefulness of the extension, we show how backward reachability analysis for *constrained dynamic pushdown networks* (see Bouajjani et al. [3]) can be encoded into a class of finite sets of linear Horn clauses for which our normalization procedure terminates after at most exponentially many steps.

1 Introduction

Horn clauses are a convenient formalism for expressing analyses of concurrent programs. Horn clauses have, e.g., successfully been applied to the analysis of cryptographic protocols [1] based on the Dolev-Yao model [4]. This model represents messages as first-order terms, where an all-powerful adversary can intercept all messages sent during the protocol's execution, delete or replace them with other messages, and generate new messages. Horn clauses provide a convenient mechanism for describing an over-approximation of the set of messages that can be learned by the adversary after arbitrary number of executions of such a protocol between different sets of participants.

Nielson et al. [12] introduced the class \mathcal{H}_1 of Horn clauses in order to model reachability in the spi-calculus. They showed that finite sets of \mathcal{H}_1 Horn clauses can be converted into equivalent tree automata in exponential time. Finite sets of \mathcal{H}_1 Horn clauses thus define regular tree languages. Moreover, they showed that the satisfiability problem for finite sets of \mathcal{H}_1 Horn clauses is DEXPTIME-hard. Independent of this work Weidenbach [13] showed earlier that the satisfiability problem for finite sets of \mathcal{H}_1 Horn clauses is decidable. The *sort resolution procedure* of Weidenbach [13] has a double exponential worst-case time-complexity and is therefore not optimal [7]. Based on the work of Nielson et al. [12], Goubault-Larrecq [7] showed how to check satisfiability of finite sets of \mathcal{H}_1 Horn clauses in exponential time by using *ordered resolution with*

* VERIMAG is a joint laboratory of CNRS, Université Joseph Fourier and Grenoble INP.

selection. Later, Goubault-Larrecq and Parrennes [8] applied \mathcal{H}_1 Horn clauses for analyzing implementations of cryptographic protocols in C.

In order to extend the class \mathcal{H}_1 , Nielsen et al. [11] introduced the *Iterative Specialization Schema*. The idea is to approximate a finite set of arbitrary Horn clauses by means of \mathcal{H}_1 Horn clauses to detect components of predicates which are *finite*. This information then is exploited to obtain an equivalent, but hopefully syntactically simpler set of Horn clauses via *instantiation*. Using these techniques, Nielsen et al. [11] successfully validated the non-trivial Yahalom protocol for key-distribution.

In the present paper, we extend \mathcal{H}_1 into another direction. We consider arbitrary *linear* Horn clauses and present a normalization semi-procedure for computing least models of finite sets of linear Horn clauses. Our normalization semi-procedure is a simplification of the procedure of Nielson et al. [12], since no pre-computations are necessary. It destructs complex heads on the fly. Moreover, it is a direct and strict extension: It always terminates on finite sets of \mathcal{H}_1 Horn clauses in at most exponentially many steps. Although satisfiability for finite sets of linear Horn clauses is undecidable [6], our semi-procedure at least terminates for certain meaningful subclasses of the class of all finite sets of linear Horn clauses. One meaningful example, where the semi-procedure terminates after at most exponentially many steps, are the finite sets of linear Horn clauses that can be used for encoding a backward reachability analysis for *constrained dynamic pushdown networks* (see Bouajjani et al. [3]). This application in particular demonstrates that potential applications are not restricted to the analysis of cryptographic protocols.

Outline. The present paper is organized as follows: We present our technical contribution — the normalization semi-procedure for finite sets of linear Horn clauses — in section 2. In section 3, we introduce an application which is not related to the analysis of cryptographic protocols, namely, backward reachability for constrained dynamic pushdown networks. We conclude in section 4. Omitted proofs can be found in the corresponding technical report.

2 Normalization of Finite Sets of Linear Horn Clauses

2.1 Basics

For a set Σ of ranked function symbols and a set \mathcal{X} of variables, $T(\Sigma, \mathcal{X})$ denotes the set of terms built up from symbols in Σ and variables in \mathcal{X} . We assume that there are k -ary function symbols ϵ_k for all $k \in \mathbb{N} \setminus \{1\}$ in Σ . The set $T(\Sigma, \emptyset)$ is also denoted by $T(\Sigma)$ and contains all ground terms, i.e. all terms without variables. We denote function symbols in the following by f, g and h , nullary function symbols (also called *constants*) by a, b and c , variables by x, y and z , and finally terms by s, t and u . *Positions* in terms are defined as usual as sequences of strictly positive integers. For a term t , t_l is the subterm of t at position l , and $t[s]_l$ the result of replacing that subterm by s . Given a set \mathbb{P} of unary predicate symbols we are able to build *atoms*. For a unary predicate $P \in \mathbb{P}$ and a term $t \in T(\Sigma, \mathcal{X})$, $P(t)$ is called an *atom*. In order to simplify notations, we also denote the atom $P(\epsilon_k(t_1, \dots, t_k))$ by $P(t_1, \dots, t_k)$ for all $k \in \mathbb{N} \setminus \{1\}$. We identify the pair (P, ϵ_k) with the k -ary predicate P . *Substitutions* are functions

$\sigma : \mathcal{X} \rightarrow T(\Sigma, \mathcal{X})$. For a term, atom or literal M , the result of applying σ to M is denoted by $M\sigma$. A substitution σ is called *ground* iff $x\sigma$ is ground for all $x \in \mathcal{X}$. We write $\{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$ to denote a substitution σ such that $x_i\sigma = t_i$ for $1 \leq i \leq n$ and $x\sigma = x$ for $x \notin \{x_1, \dots, x_n\}$.

A *clause* is a finite set of *positive literals* A and *negative literals* $\neg A$, where A is an *atom*. We also denote a *clause* $\{L_1, \dots, L_k\}$ by $L_1 \vee \dots \vee L_k$, where L_1, \dots, L_k are literals. For a term t we denote the set of variables occurring in t by $\mathbf{Vars}(t)$. Respectively, we denote this set by $\mathbf{Vars}(L)$ for a literal L and $\mathbf{Vars}(\mathcal{C})$ for a clause \mathcal{C} .

A *Horn clause* is a clause that contains at most one positive literal. A Horn clause $A \vee \neg A_1 \vee \dots \vee \neg A_k$ is also written as $A \Leftarrow A_1 \wedge \dots \wedge A_k$. A is the *head* and $A_1 \wedge \dots \wedge A_k$ is the *body* of this Horn clause. Again we define $\mathbf{Vars}(A_1 \wedge \dots \wedge A_k) := \bigcup_{i=1}^k \mathbf{Vars}(A_i)$. If $k = 0$ holds, then the body is denoted by ϵ . A negative literal within a Horn clause is also called *query*. An *interpretation* is a set of ground atoms. A *model* of a set of clauses is an interpretation, in which all clauses are satisfied. We also consider an interpretation as a mapping from predicates to sets of ground terms, i.e., for some interpretation \mathcal{I} we write $t \in \mathcal{I}(P)$ iff $P(t) \in \mathcal{I}$ holds. For every set S of Horn clauses there exists a unique least model, which we denote by \mathcal{H}_S . \mathcal{H}_S is the set of ground atoms which can be derived using the rule

$$\frac{A_1\sigma \dots A_k\sigma}{A\sigma} \sigma \text{ is ground and } A \Leftarrow A_1 \wedge \dots \wedge A_k \in S.$$

Hence we can associate a *derivation* with every atom in \mathcal{H}_S . The *size* of this derivation is the number of times the above rule is applied.

A term, atom or literal is called *linear* iff no variable occurs twice in it. A Horn clause $H \Leftarrow \mathcal{B}$ is called *linear* iff its head H is linear. A linear Horn clause \mathcal{C} is called *normal* iff it is of the form

$$P(f(x_1, \dots, x_k)) \Leftarrow Q_1(x_1) \wedge \dots \wedge Q_k(x_k).$$

This is equivalent to the definition of Nielson et al. [12]. We denote the set of all normal Horn clauses by \mathcal{N} . A normal Horn clause can be considered as a transition rule in a tree automaton. The predicates are the states of the tree automaton. Accordingly, the least model \mathcal{H}_S of a finite set S of normal Horn clauses maps every occurring predicate to a regular tree language. A set S of Horn clauses is called *normalizable* iff there exists a finite set S' of normal Horn clauses which is equivalent to S (up to auxiliary predicates).

Two variables x and y are *connected* in a clause $\{L_1, \dots, L_n\}$ iff $x \cong y$, where \cong is the smallest equivalence relation such that the following holds: if x and y occur in the same literal L_i for $i \in \{1, \dots, n\}$, then $x \cong y$. Two terms t_1 and t_2 are *connected* in a clause \mathcal{C} iff there exist variables $x_1 \in \mathbf{Vars}(t_1)$ and $x_2 \in \mathbf{Vars}(t_2)$ such that x_1 and x_2 are connected in \mathcal{C} . Two (occurrences of) terms are called *siblings* in a term or literal iff they occur as an argument of a common father. A Horn clause $H \Leftarrow \mathcal{B}$ is called \mathcal{H}_1 iff it is linear and the following statement holds: if two variables x, y occur in H and are connected in \mathcal{B} , then x and y are siblings in H (cf. Nielson et al. [12]).

Only regular tree languages can be expressed using finite sets of \mathcal{H}_1 Horn clauses. Nielson et al. [12] have shown this by specifying a normalization procedure, which, given a finite set of \mathcal{H}_1 Horn clauses, constructs an (up to auxiliary predicates) equivalent finite set of normal Horn clauses. Using finite sets of *linear* Horn clauses we

are also able to encode non-regular tree languages. The least model of the finite set $S = \{P(f(x), f(y)) \Leftarrow P(x, y), P(a, a) \Leftarrow \epsilon\}$ of linear Horn clauses, for instance, is given by $\mathcal{H}_S = \{P(a, a), P(f(a), f(a)), P(f(f(a)), f(f(a))), \dots\}$. Obviously, the set \mathcal{H}_S is not regular. Therefore, it is not possible to construct an equivalent finite set of normal Horn clauses for every finite set of linear Horn clauses. Accordingly, it is not the case that the normalization semi-procedure we present in the present article will terminate for every finite set of linear Horn clauses. Nevertheless there exist meaningful examples of finite sets of linear Horn clauses that are not finite sets of \mathcal{H}_1 Horn clauses which still define regular tree languages. Our method moreover can be considered as an instantiation technique (cf. Nielsen et al. [11]).

We emphasize that we cannot expect an algorithm that terminates for every finite set of linear Horn clauses, because the satisfiability problem for finite sets of linear Horn clauses is undecidable Goubault-Larrecq [6].

2.2 The Normalization Semi-procedure

The goal of our normalization semi-procedure we are now going to present is to construct an equivalent set of *normal* Horn clauses from a given set of *linear* Horn clauses. Our normalization semi-procedure is a strict extension and at the same time a simplification of the normalization procedure of Nielson et al. [12] for finite sets of \mathcal{H}_1 Horn clauses. The main difference is that we introduce auxiliary predicates on demand during the normalization process instead of doing so beforehand.

In order to describe our normalization semi-procedure, we consider sets of unary predicates as unary predicates. Additionally, we identify the set $\{P\}$ with the predicate P . A predicate $\{P_1, \dots, P_n\}$ stands for the intersection of the success sets of the contained predicates P_1, \dots, P_n . We therefore call them *intersection-predicates*. Conceptually an intersection predicate $\{P_1, \dots, P_n\}$ is defined by the Horn clause

$$\{P_1, \dots, P_n\}(X) \Leftarrow P_1(X) \wedge \dots \wedge P_n(X).$$

The additional power of our normalization semi-procedure relies on the fact that it introduces auxiliary predicates of the form P_t , where P is a predicate and t is a term that is built up from constants, variables, function symbols, unary predicates and a special constant \square . The unary predicates are used as constants. Predicates of the form P_t are called *push-predicates*. An example of a push-predicate is the predicate $P_{(f(\square), Q)}$, where f is a unary function symbol and P, Q are predicates. In our normalization semi-procedure, the term t describes the *context* in which the auxiliary predicate P_t is introduced.

We define a relation \triangleright on sets of Horn clauses. The relation \triangleright represents one step of the saturation process. If this saturation process terminates, then the set of *normal* Horn clauses that are contained in the saturated set of Horn clauses is (up to auxiliary predicates) equivalent to the initial set of Horn clauses (*soundness*). For convenience we assume w.l.o.g. that all clauses $H \Leftarrow B$ in the initial set satisfy the property $\text{Vars}(H) \subseteq \text{Vars}(B)$. This can be done w.l.o.g., because, if some variable x occurred in H but not in B , then we could replace the clause with $H \Leftarrow B \wedge P_{\text{all}}(x)$ where P_{all} is an auxiliary predicate which accepts all terms. This property is also satisfied by all new clauses generated by our normalization semi-procedure.

In the following x_j are mutually distinct variables. The relation \triangleright over sets of Horn clauses is specified by an inference system. The inference rules are of the form

$$\frac{P_1 \cdots P_m}{C_1 \cdots C_n},$$

where P_1, \dots, P_m are the premises and C_1, \dots, C_n are the conclusions. The relation \triangleright is defined as follows: $S \triangleright S \cup \{C_1, \dots, C_n\}$ iff the inference rule

$$\frac{P_1 \cdots P_m}{C_1 \cdots C_n}$$

is a member of the inference system, $P_1, \dots, P_m \in S$, and $\{C_1, \dots, C_n\} \not\subseteq S$. A set S of Horn clauses is called *saturated* iff $\nexists S' : S \triangleright S'$. Our inference system is defined through the following inference rules:

$$\frac{P(x) \Leftarrow Q(x) \quad Q(f(x_1, \dots, x_k)) \Leftarrow \bigwedge_{i=1}^k Q_i(x_i)}{P(f(x_1, \dots, x_k)) \Leftarrow \bigwedge_{i=1}^k Q_i(x_i)} \text{ (cp)}$$

$$\frac{P(t) \Leftarrow \bigwedge_{i=1}^k Q_i(x_i)}{P(t[x]_l) \Leftarrow P_{t[\square]_l\{x_i \mapsto Q_i | i=1, \dots, k\}}(x) \wedge \bigwedge_{i \in \{1, \dots, k | x_i \notin \mathbf{Vars}(t)_l\}} Q_i(x_i)} \text{ (sp)}$$

$$P_{t[\square]_l\{x_i \mapsto Q_i | i=1, \dots, k\}}(t|l) \Leftarrow \bigwedge_{i \in \{1, \dots, k | x_i \in \mathbf{Vars}(t)_l\}} Q_i(x_i)$$

Here, l is a non-root position in t , $t|_l$ is not a variable, $\{x_1, \dots, x_k\} \subseteq \mathbf{Vars}(t)$, $x \notin \mathbf{Vars}(t)$, and $\mathcal{H}_{S \cap \mathcal{N}}(Q_i) \neq \emptyset$ for all $i = 1, \dots, k$. Recall that $\mathcal{H}_{S \cap \mathcal{N}}(Q_i) \neq \emptyset$ means that there exists a term which satisfies the predicate Q_i in the least model of the normal Horn clauses contained in S .

$$\frac{H \Leftarrow \mathcal{B} \wedge Q(f(t_1, \dots, t_k)) \quad Q(f(x_1, \dots, x_k)) \Leftarrow \bigwedge_{i=1}^k Q_i(x_i)}{H \Leftarrow \mathcal{B} \wedge \bigwedge_{i=1}^k Q_i(t_i)} \text{ (cut)}$$

$$\frac{H \Leftarrow \mathcal{B} \wedge \bigwedge_{i=1}^k Q_i(x)}{H \Leftarrow \mathcal{B} \wedge (\bigcup_{i=1}^k Q_i)(x)} \text{ (cap1)}$$

Here, $k > 1$, and $x \notin \mathbf{Vars}(\mathcal{B})$.

$$\frac{\overbrace{\{P_i\}(f(x_1, \dots, x_k)) \Leftarrow \bigwedge_{j=1}^k Q_{i,j}(x_j)}^{i=1, \dots, n} \quad H \Leftarrow \mathcal{B} \wedge \{P_1, \dots, P_n\}(x)}{\{P_1, \dots, P_n\}(f(x_1, \dots, x_k)) \Leftarrow \bigwedge_{j=1, \dots, k} (Q_{1,j} \cup \dots \cup Q_{n,j})(x_j)} \text{ (cap2),}$$

Here, $n > 1$, and $x \notin \mathbf{Vars}(\mathcal{B})$.

$$\frac{H \Leftarrow \mathcal{B} \wedge Q(x)}{H \Leftarrow \mathcal{B}} \text{ (elim)}$$

Here, $\mathcal{H}_{S \cap \mathcal{N}}(Q) \neq \emptyset$, and $x \notin \mathbf{Vars}(H) \cup \mathbf{Vars}(\mathcal{B})$.

In order to apply the rule (**elim**) or the rule (**sp**), we have to check, whether or not $\mathcal{H}_{S \cap \mathcal{N}}(Q) \neq \emptyset$ holds. This means, we have to check emptiness for tree automata. This can be done in polynomial time. During the execution of our normalization semi-procedure, the emptiness-information can be computed on the fly. This means: For each predicate P , we maintain the information, whether or not $\mathcal{H}_{S \cap \mathcal{N}}(P) \neq \emptyset$ holds. Whenever we add a normal Horn clause, we update this information.

The substantial difference between our normalization semi-procedure and the normalization procedure of Nielson et al. [12] is the rule (**sp**). The rule (**sp**) dispenses with the need of the pre-processing phase as described by Nielson et al. [12] or by Goubault-Larrecq [7]. The pre-processing phases of Nielson et al. [12] and Goubault-Larrecq [7] essentially decompose complex heads of Horn clauses $P(t) \Leftarrow \mathcal{B}$ through the introduction of auxiliary predicates. At the end of these pre-processing phases all heads are of the form $P(x_1, \dots, x_k)$ or $P(f(x_1, \dots, x_k))$. The Horn clause $P(f(x), y) \Leftarrow Q_1(x) \wedge Q_2(y)$, for instance, is replaced by the Horn clauses $P(x, y) \Leftarrow Q(x) \wedge Q_2(y)$, $Q(f(x)) \Leftarrow Q_1(x)$, where Q is a fresh auxiliary predicate. Note that this step is done by the rule (**sp**). However, the Horn-clause $P(f(x), y) \Leftarrow Q(x, y)$ — which is not \mathcal{H}_1 — cannot be replaced by (up to auxiliary predicates) equivalent Horn-clauses whose heads are of the form $P(x_1, \dots, x_k)$ or $P(f(x_1, \dots, x_k))$ during a pre-processing phase, since the variables x and y are connected in the body. Here, the observation is that connection between x and y will be eliminated during the normalization process. Therefore we postpone the decomposition of complex heads until the connection between the variables in the bodies are eliminated by the normalization process. The rule (**sp**) does this together with a clever naming of the introduced auxiliary predicates (the push-predicates). In consequence, using our new inference rules, we can normalize finite sets of linear Horn clauses that can neither be solved through the normalization procedure of Nielson et al. [12] nor through the ordered resolution procedure of Goubault-Larrecq [7].

Example 1. We consider the following finite set S of linear Horn clauses:

$$P(f(x_1), x_2) \Leftarrow P(x_1, x_2) \quad (1)$$

$$P(x_1, x_2) \Leftarrow Q(x_1) \wedge R(x_2) \quad (2)$$

$$Q(a) \Leftarrow \epsilon \quad (3)$$

$$R(b) \Leftarrow \epsilon \quad (4)$$

$$R(g(x)) \Leftarrow R(x) \quad (5)$$

We have $\mathcal{H}_S(P) = \{(f^i(a), g^j(b)) \mid i, j \in \mathbb{N}\}$. Our normalization semi-procedure performs the following steps:

$$\text{(cut) (1) (2)} : \quad P(f(x_1), x_2) \Leftarrow Q(x_1) \wedge R(x_2) \quad (6)$$

$$\text{(sp) (6)} : \quad P(x, x_2) \Leftarrow P_{(\square, R)}(x) \wedge R(x_2) \quad (7)$$

$$P_{(\square, R)}(f(x_1)) \Leftarrow Q(x_1) \quad (8)$$

$$\text{(cut) (1) (7)} : \quad P(f(x_1), x_2) \Leftarrow P_{(\square, R)}(x_1) \wedge R(x_2) \quad (9)$$

$$\text{(sp) (9)} : \quad P_{(\square, R)}(f(x_1)) \Leftarrow P_{(\square, R)}(x_1) \quad (10)$$

Observe that the last normalization step closes an important cycle due to the naming of the introduced auxiliary predicates. The resulting finite set S' of linear Horn clauses is saturated and contains the following *normal* Horn clauses:

$$\begin{array}{ll}
P(x_1, x_2) \Leftarrow Q(x_1) \wedge R(x_2) & Q(a) \Leftarrow \epsilon \\
R(b) \Leftarrow \epsilon & R(g(x)) \Leftarrow R(x) \\
P(x, x_2) \Leftarrow P_{(\square, R)}(x) \wedge R(x_2) & P_{(\square, R)}(f(x_1)) \Leftarrow Q(x_1) \\
P_{(\square, R)}(f(x_1)) \Leftarrow P_{(\square, R)}(x_1) &
\end{array}$$

The set of these normal Horn clauses is (up to auxiliary predicates) equivalent to the initial set of *linear* Horn clauses. In particular, we have

$$\mathcal{H}_{S' \cap \mathcal{N}}(P) = \{(f^i(a), g^j(b)) \mid i, j \in \mathbb{N}\} = \mathcal{H}_S(P).$$

Note that this result cannot be obtained by using the instantiation techniques of Nielsen et al. [11]. \square

The set of derivable facts is (up to auxiliary predicates) preserved by executing normalization steps:

Lemma 1 (Soundness). *Let S be a set of linear Horn clauses and $S \triangleright^* S'$. Then $\mathcal{H}_{S'}(P) = \mathcal{H}_S(P)$ holds for every predicate P which occurs in S .*

Proof. Let us fix some sequence $S = S_0 \triangleright \dots \triangleright S_M = S'$. We set $I_K := S_K \setminus S_{K-1}$ for all $K = 1, \dots, M$, and $I := S' \setminus S$. We define a size function Size which maps a derivation in S' to an element from \mathbb{N}^M by:

$$\begin{aligned}
\text{Size} \left(\frac{D_1 \cdots D_k}{A} C \right) &:= \text{Size}(C) + \sum_{i=1}^k \text{Size}(D_i) \\
\text{Size}(C) &:= \begin{cases} (\delta_{1j}, \dots, \delta_{Mj}) & \text{if } C \in I_j \\ (0, \dots, 0) & \text{otherwise} \end{cases}
\end{aligned}$$

Here, δ denotes the Kronecker-Delta, i.e., δ_{ij} equals 1 if $i = j$ and 0 otherwise. The elements of \mathbb{N}^M are ordered lexicographically with reversed significance, i.e., $(a_1, \dots, a_M) < (b_1, \dots, b_M)$ holds iff there exists some $j \in \{1, \dots, M\}$ such that $a_j < b_j$ and $a_i = b_i$ for all $i > j$ holds. Note that the following holds: If we replace a subderivation of a derivation with a smaller derivation, then the derivation itself gets smaller. Furthermore, if $\text{Size}(D) = (0, \dots, 0)$ holds for a derivation D , then only clauses from S are used in D .

Let \overline{D} be a derivation for a ground atom $\overline{P}(\overline{t})$, where \overline{P} is a predicate from S and $\text{Size}(\overline{D}) > (0, \dots, 0)$ holds. It is sufficient to show, that there exists a derivation D for $\overline{P}(\overline{t})$ with $\text{Size}(D) < \text{Size}(\overline{D})$.

There must be a subderivation D' of \overline{D} , where the last inference is an application of a Horn clause from I_K for some $K \in \{1, \dots, M\}$.

Case 1: The normalization step from S_{K-1} to S_K is an application of the inference rule **(cp)**. Let $P(f(x_1, \dots, x_k)) \Leftarrow \bigwedge_{i=1}^k Q(x_i)$ be the Horn clause in I_K . There

are Horn clauses $P(x) \Leftarrow Q(x)$ and $Q(f(x_1, \dots, x_k)) \Leftarrow \bigwedge_{i=1}^k Q(x_i)$ in S_{K-1} . Thus we have

$$D' = \frac{D_1 \cdots D_k}{P(f(t_1, \dots, t_k))} P(f(x_1, \dots, x_k)) \Leftarrow \bigwedge_{i=1}^k Q(x_i),$$

where the conclusion of D_i is $Q_i(t_i)$ for all $i = 1, \dots, k$. In order to get a smaller derivation, we substitute the subderivation D' with the following smaller derivation

$$\frac{\frac{D_1 \cdots D_k}{Q(f(t_1, \dots, t_k))} Q(f(x_1, \dots, x_k)) \Leftarrow \bigwedge_{i=1}^k Q(x_i)}{P(f(t_1, \dots, t_k))} P(x) \Leftarrow Q(x)$$

Case 2: The normalization step from S_{K-1} to S_K is an application of the inference rule **(sp)**. Thus we have $\{\mathcal{C}_2\} \subseteq I_K \subseteq \{\mathcal{C}_1, \mathcal{C}_2\}$, where

$$\begin{aligned} \mathcal{C}_1 &= P(t[x]_l) \Leftarrow P_{t[\square]_l \{x_i \mapsto Q_i \mid i=1, \dots, k\}}(x) \wedge \bigwedge_{i \in \{1, \dots, k \mid x_i \notin \mathbf{Vars}(t_l)\}} Q_i(x_i) \\ \mathcal{C}_2 &= P_{t[\square]_l \{x_i \mapsto Q_i \mid i=1, \dots, k\}}(t_l) \Leftarrow \bigwedge_{i \in \{1, \dots, k \mid x_i \in \mathbf{Vars}(t_l)\}} Q_i(x_i). \end{aligned}$$

Case 2.1: The last inference of D' is an application of \mathcal{C}_1 . Thus D' has the form

$$\frac{D'' \cdots}{P(\cdots)} \mathcal{C}_1,$$

where the conclusion of D'' is $P_{t[\square]_l \{x_i \mapsto Q_i \mid i=1, \dots, k\}}(t'')$. The last inference of the derivation D'' must be some application of a Horn clause from I . Now we replace the sub-derivation D'' instead of the sub-derivation D' .

Case 2.2: The last inference of D' is an application of \mathcal{C}_2 .

Case 2.2.1: There is a sub-derivation of \overline{D} of the form

$$\frac{\frac{D_{b_1} \cdots D_{b_r}}{P_{t[\square]_l \{x_i \mapsto Q_i \mid i=1, \dots, k\}}(t_l \{x_i \mapsto t_i \mid i = 1, \dots, k\})} \mathcal{C}_2}{P(t\{x_i \mapsto t_i \mid i = 1, \dots, k\})} D_{a_1} \cdots D_{a_s} \mathcal{C}_1,$$

where the conclusion of the derivation D_i is $Q_i(t_i)$ for all $i = 1, \dots, k$ and moreover $\{a_1, \dots, a_s\} \cup \{b_1, \dots, b_r\} = \{1, \dots, k\}$, $a_1, \dots, a_s, b_1, \dots, b_r$ are pair-wise distinct, $x_{a_1}, \dots, x_{a_s} \notin \mathbf{Vars}(t_l)$, and $x_{b_1}, \dots, x_{b_r} \in \mathbf{Vars}(t_l)$. Since the Horn clause $\mathcal{C}' = P(t) \Leftarrow \bigwedge_{i=1}^k Q_i(x_i)$ must be a member of S_{K-1} , we can replace the derivation D' with the smaller derivation

$$\frac{D_1 \cdots D_k}{P(t\{x_i \mapsto t_i \mid i = 1, \dots, k\})} \mathcal{C}'.$$

Case 2.2.2: There is no sub-derivation of \overline{D} of the form

$$\frac{\frac{D_{b_1} \cdots D_{b_r}}{P_{t[\square]_l \{x_i \mapsto Q_i \mid i=1, \dots, k\}}(t_l \{x_i \mapsto t_i \mid i = 1, \dots, k\})} \mathcal{C}_2}{P(t\{x_i \mapsto t_i \mid i = 1, \dots, k\})} D_{a_1} \cdots D_{a_s} \mathcal{C}_1,$$

where the conclusion of the derivation D_i is $Q_i(t_i)$ for all $i = 1, \dots, k$ and moreover $\{a_1, \dots, a_s\} \cup \{b_1, \dots, b_r\} = \{1, \dots, k\}$, $a_1, \dots, a_s, b_1, \dots, b_r$ are pair-wise distinct, $x_{a_1}, \dots, x_{a_s} \notin \mathbf{Vars}(t_i)$ and $x_{b_1}, \dots, x_{b_r} \in \mathbf{Vars}(t_i)$. We choose some sub-derivation D'' of the form

$$\frac{D' \dots}{\tilde{P}(\tilde{t})} \mathcal{C}.$$

The Horn clause \mathcal{C} must be from the set I . We replace the sub-derivation D'' instead of the sub-derivation D' .

The other cases, which are similar to those encountered for classes like \mathcal{H}_1 , can also be treated straightforwardly. \square

Whenever the normalization procedure terminates, the normal Horn clauses which are contained in the saturated set describe the least model completely:

Lemma 2 (Completeness). *Let S be a set of linear Horn clauses and $S \stackrel{*}{\triangleright} \overline{S}$, where \overline{S} is saturated. Then $\mathcal{H}_{\overline{S} \cap \mathcal{N}}(P) = \mathcal{H}_{\overline{S}}(P)$ holds for every predicate P which occurs in \overline{S} .*

Proof. First of all, we define the size of a derivation. For that we define the measure $\mu(C)$ of a Horn clause C as $3^m + n$, where m is the number of sub-terms occurring in the head and n is the number of sub-terms occurring in the body. Then, the measure $\mu(D)$ of a derivation D is simply the sum of the measures of the Horn clauses used within the derivation.

For the sake of contradiction assume that $\tilde{t} \in \mathcal{H}_{\overline{S}}(\overline{P}) \setminus \mathcal{H}_{\overline{S} \cap \mathcal{N}}(\overline{P})$ holds, where \overline{P} occurs in \overline{S} . Let \overline{D} be a derivation for the atom $\overline{P}(\tilde{t})$ of minimal measure. Within \overline{D} there exists some sub-derivation

$$D = \frac{D_1 \dots D_k}{P(t)} \mathcal{C}$$

such that $\mathcal{C} \notin \mathcal{N}$ and D_1, \dots, D_k are *normal* derivations, i.e., only normal Horn clauses are used within the derivations D_1, \dots, D_k .

Case 1: $\mathcal{C} = P(x) \Leftarrow Q(x)$. In this case we have $k = 1$ and

$$D_1 = \frac{D'_1 \dots D'_l}{Q(t)} Q(f(x_1, \dots, x_l)) \Leftarrow Q_1(x_1) \wedge \dots \wedge Q_l(x_l).$$

Since \overline{S} is saturated, we have $\mathcal{C}' := P(f(x_1, \dots, x_l)) \Leftarrow Q_1(x_1) \wedge \dots \wedge Q_l(x_l) \in \overline{S}$. Thus we can replace the sub-derivation D with the smaller derivation

$$\frac{D'_1 \dots D'_l}{P(t)} \mathcal{C}'.$$

Thus we have constructed a smaller derivation for $\overline{P}(\tilde{t})$ — contradiction.

Case 2: $\mathcal{C} = P(\tilde{t}) \Leftarrow \bigwedge_{i=1}^k Q_i(x_i)$, l is a non-root position in \tilde{t} , the term $\tilde{t}|_l \notin \mathbf{Vars}$ and $\{x_1, \dots, x_k\} \subseteq \mathbf{Vars}(\tilde{t})$. The conclusions of D_1, \dots, D_k are $Q_1(t_1), \dots, Q_k(t_k)$, respectively. It holds $t = \tilde{t}\{x_i \mapsto t_i \mid i = 1, \dots, k\}$. Since \overline{S} is saturated, the following Horn clauses are members of \overline{S} :

$$\begin{aligned} \mathcal{C}_1 &:= P(\tilde{t}[x]_l) \Leftarrow P_{\tilde{t}[\square]_l\{x_i \mapsto Q_i \mid i=1, \dots, k\}}(x) \wedge \bigwedge_{i \in \{1, \dots, k \mid x_i \notin \mathbf{Vars}(\tilde{t}|_l)\}} Q_i(x_i) \\ \mathcal{C}_2 &:= P_{\tilde{t}[\square]_l\{x_i \mapsto Q_i \mid i=1, \dots, k\}}(\tilde{t}|_l) \Leftarrow \bigwedge_{i \in \{1, \dots, k \mid x_i \in \mathbf{Vars}(\tilde{t}|_l)\}} Q_i(x_i) \end{aligned}$$

Here, $x \notin \mathbf{Vars}(\tilde{t})$. Let $\{a_1, \dots, a_s\} \cup \{b_1, \dots, b_r\} = \{1, \dots, k\}$ such that $a_1, \dots, a_s, b_1, \dots, b_r$ are pair-wise distinct, $x_{a_1}, \dots, x_{a_s} \notin \mathbf{Vars}(\tilde{t}|_l)$ and furthermore $x_{b_1}, \dots, x_{b_r} \in \mathbf{Vars}(\tilde{t}|_l)$. We can replace the sub-derivation D with the smaller derivation

$$\frac{\frac{D_{b_1} \cdots D_{b_r}}{P_{\tilde{t}[\square]_l\{x_i \mapsto Q_i \mid i=1, \dots, k\}}(\tilde{t}|_l\{x_i \mapsto t_i \mid i = 1, \dots, k\})} \mathcal{C}_2 \quad D_{a_1} \cdots D_{a_s} \mathcal{C}_1}{P(t)}$$

Thus we have constructed a smaller derivation for $\overline{P}(\tilde{t})$ — contradiction.

The other cases, which are similar to those encountered for classes like \mathcal{H}_1 , can also be treated straightforwardly. \square

Lemma 1 and Lemma 2 finally imply our main theorem:

Theorem 1. *Let S and S' be sets of linear Horn clauses such that $S \stackrel{*}{\triangleright} S'$ holds and S' is saturated. Then $\mathcal{H}_S(P) = \mathcal{H}_{S' \cap \mathcal{N}}(P)$ holds and thus $\mathcal{H}_S(P)$ is tree regular for every predicate P which occurs in S , i.e., the least model is tree regular.* \square

Because of the above theorem, our normalization semi-procedure does not terminate, if one applies it to a finite set of Horn clauses whose least model is not tree regular. The least model of the finite set

$$S = \{P(a, a) \Leftarrow \epsilon, P(f(x), f(y)) \Leftarrow P(x, y)\}$$

of linear Horn clauses, for instance, is $\mathcal{H}_S = \{P(f^i(a), f^i(a)) \mid i \in \mathbb{N}\}$ and thus not tree regular. There are nonetheless finite sets of linear Horn clauses, whose least model is tree regular, while our normalization semi-procedure still does not terminate. It will for instance not terminate on the set

$$S' := S \cup \{P(f(x), y) \Leftarrow P(x, y), P(x, f(y)) \Leftarrow P(x, y)\}$$

of linear Horn clauses, since S' is a superset of S . Nonetheless, the least model $H_{S'} = \{P(f^i(a), f^j(a)) \mid i, j \in \mathbb{N}\}$ is tree regular.

Our normalization semi-procedure can be improved in several directions. We can, for instance, augment our normalization semi-procedure with *subsumption*, i.e., in each normalization step we can delete *subsumed* Horn clauses. For instance, the Horn clause $H \Leftarrow B \wedge Q(x)$ is subsumed, after we apply the rule (**elim**). In the present paper we

will not discuss these improvements in detail. Instead, we remark that our normalization semi-procedure combines the two phases of the normalization procedure given in [12]. Using the same arguments as in [12], we deduce that for a given finite set of \mathcal{H}_1 Horn clauses, our normalization procedure terminates after at most exponentially many normalization steps:

Corollary 1. *Our normalization semi-procedure normalizes a finite set of \mathcal{H}_1 Horn clauses in exponential time.* \square

2.3 Instantiation

In this section we compare our techniques with instantiation as described by Nielsen et al. [11]. Assume that S is a finite set of Horn clauses. Let us consider a Horn clause $C \equiv H \Leftarrow B \in S$ and a set $\{x_1, \dots, x_k\} \subseteq \mathbf{Vars}(B)$ of variables. If the set

$$I := \{(t_1, \dots, t_k) \mid t_1, \dots, t_k \in T(\Sigma), \mathcal{H}_S \models B\{x_1 \mapsto t_1, \dots, x_k \mapsto t_k\}\}$$

of possible values for the vector of variables (x_1, \dots, x_k) is *finite*, we are able to *instantiate* this vector of variables with all possible values. This means we can replace the Horn clause C by the Horn clauses $C\{x_1 \mapsto t_1, \dots, x_k \mapsto t_k\}$ for all $(t_1, \dots, t_k) \in I'$, where I' is some finite superset of I . We obtain a finite set S' of Horn clauses that is equivalent to S .

Thus, in some cases we can replace a Horn Clause C that does not belong to \mathcal{H}_1 (or to some other syntactically defined class) by a set of Horn clauses which have fewer occurrences of variables and thus may belong to \mathcal{H}_1 . We refer to Nielsen et al. [11] for a detailed explanation of this framework.

Example 2. Consider the following finite set S of linear Horn clauses:

$$P(x, y) \Leftarrow A(x) \wedge B(y) \tag{11}$$

$$A(a) \Leftarrow \epsilon \tag{12}$$

$$B(b) \Leftarrow \epsilon \tag{13}$$

$$B(h(x)) \Leftarrow B(x) \tag{14}$$

$$Q(a) \Leftarrow \epsilon \tag{15}$$

$$Q(b) \Leftarrow \epsilon \tag{16}$$

$$P(f(x), g(y)) \Leftarrow P(x, y) \wedge Q(x) \tag{17}$$

The Horn clause (17) is not \mathcal{H}_1 , since x and y are connected in the body, but they are not siblings in the head. However, the set $\mathcal{H}_S(Q) = \{a, b\}$ is finite. Therefore, the variable x in the Horn clause (17) can be instantiated, i.e., (17) can be replaced by

$$P(f(a), g(y)) \Leftarrow P(a, y) \wedge Q(a) \tag{18}$$

$$P(f(b), g(y)) \Leftarrow P(b, y) \wedge Q(b). \tag{19}$$

By doing so, a finite set S' of Horn clauses is obtained that is equivalent to S , i.e. $\mathcal{H}_{S'} = \mathcal{H}_S$. Since all Horn clauses of S' belong to the class \mathcal{H}_1 , now the methods of Nielson et al. [12] or the methods of Goubault-Larrecq [7] can be applied.

Since all clauses of S are linear, our normalization semi-procedure could also be applied directly. The algorithm performs the following steps:

$$\text{(cut)} \text{ (17) (11)} \quad P(f(x), g(y)) \Leftarrow A(x) \wedge B(y) \wedge Q(x) \quad (20)$$

$$\text{(cap1)} \text{ (20)} \quad P(f(x), g(y)) \Leftarrow \{A, Q\}(x) \wedge B(y) \quad (21)$$

$$\text{(cap2)} \text{ (12) (15) (21)} \quad \{A, Q\}(a) \Leftarrow \epsilon \quad (22)$$

$$\text{(sp)} \text{ (21)} \quad P(x, g(y)) \Leftarrow P_{(\square, g(B))}(x) \wedge B(y) \quad (23)$$

$$P_{(\square, g(B))}(f(x)) \Leftarrow \{A, Q\}(x) \quad (24)$$

$$\text{(sp)} \text{ (23)} \quad P(x, y) \Leftarrow P_{(\square, g(B))}(x) \wedge P_{(P_{(\square, g(B)), \square})}(y) \quad (25)$$

$$P_{(P_{(\square, g(B)), \square})}(g(y)) \Leftarrow B(y) \quad (26)$$

$$\text{(cut)} \text{ (17) (25)} \quad P(f(x), g(y)) \Leftarrow P_{(\square, g(B))}(x) \wedge P_{(P_{(\square, g(B)), \square})}(y) \wedge Q(x) \quad (27)$$

$$\text{(cap1)} \text{ (27)} \quad P(f(x), g(y)) \Leftarrow \{P_{(\square, g(B))}, Q\}(x), P_{(P_{(\square, g(B)), \square})}(y) \quad (28)$$

Our normalization procedure terminates, because $\mathcal{H}_{S \cap \mathcal{N}}(\{P_{(\square, g(B))}, Q\}) = \emptyset$. In fact it always terminates, if the predicate Q is defined by finitely many Horn clauses of the form $Q(t) \Leftarrow \epsilon$. This is one situation where instantiation can be applied. \square

Our example shows that our normalization procedure in some cases dispenses with the need for an instantiation pre-processing step. In example [11](#) we have seen that there are cases where instantiation does not work, but our techniques can be applied. On the other hand, there also exist examples where instantiation can be applied such that the resulting finite set of Horn clauses is a finite set of \mathcal{H}_1 Horn clauses, but our normalization semi-procedure does not terminate. This means that our techniques neither is subsumed by instantiation, nor does dispense with the need of instantiation in all cases. Instead, our normalization semi-procedure can be enhanced with instantiation in the same way as \mathcal{H}_1 solving can be enhanced.

3 An Application: Backward Reachability Analysis for Constrained Dynamic Pushdown Networks

Bouajjani et al. [\[3\]](#) introduced *constrained dynamic pushdown networks (CDPNs)* as a model of recursive programs with dynamic thread creation. CDPNs extend the modeling power of *pushdown automata* [\[2, 5, 9\]](#), and even of PADs [\[10\]](#).

Assume that we are given a regular set C of *bad configurations*. *Backward reachability analysis* tries to determine the set $\text{pre}^*(C)$ of all configurations from which bad configurations can be reached. The system then can be considered as safe if the intersection of $\text{pre}^*(C)$ with the set of initial configurations is empty. In this section, we demonstrate that backward reachability can be described by means of finite sets of linear Horn clauses which our normalization semi-procedure normalizes in exponential time in the worst-case.

A *constrained dynamic pushdown network* (CDPN for short) is a tuple $M = (P, \Gamma, \Delta)$, where P is a finite set of *control states*, Γ is a finite set of stack symbols disjoint

from P , and Δ is a finite set of transition rules of the following forms: either (a) $\Phi : p\gamma \hookrightarrow p_1w_1$, or (b) $\Phi : p\gamma \hookrightarrow p_1w_1 \triangleright p_2w_2$, where $p, p_1, p_2 \in P$, $\gamma \in \Gamma$, $w_1, w_2 \in \Gamma^*$, and Φ is a *constraint over* P , i.e., a regular word language over P .

For the remainder of this section let M be a CDPN. In order to define the set of all *configurations* for M , we consider a stack symbol $\gamma \in \Gamma$ as a unary function symbol and a control state $p \in P$ as a 2-ary function symbol. We denote the empty stack by the constant $\mathbf{0}$ and use the binary function symbol \mathbf{cons} and the constant \mathbf{nil} to construct lists. For a unary function symbol γ and a term t we also write γt instead of $\gamma(t)$. In contrast to Bouajjani et al. [3] we define the set Proc_M of all configurations for M by:

$$\begin{aligned} \text{Stack} &::= \Gamma(\text{Stack}) \mid \mathbf{0} & \text{Children} &::= \mathbf{cons}(\text{Proc}_M, \text{Children}) \mid \mathbf{nil} \\ \text{Proc}_M &::= P(\text{Stack}, \text{Children}) \end{aligned}$$

We also denote the function symbol \mathbf{cons} by the right associative infix functional symbol $::$ and the constant \mathbf{nil} by $[]$. Furthermore, we abbreviate the list $t_1 :: \dots :: t_n :: []$ as $[t_1, \dots, t_n]$. We denote the control state p of a process $t = p(s, [t_1, \dots, t_n])$ by $\text{st}(t)$. A process $p(\gamma_1 \dots \gamma_k \mathbf{0}, [t_1, \dots, t_n])$ consists of the control state p , the stack $\gamma_1 \dots \gamma_k \mathbf{0}$ (γ_1 is the top-most stack-element), and a list of child processes $[t_1, \dots, t_n]$. $p(\gamma \delta \mathbf{0}, [q(\mathbf{0}, [])])$, for instance, is a configuration that consists of one root process and one child process.

The transition relation \rightarrow_M for a CDPN M is the smallest relation between configurations that fulfills the following properties:

1. If $\Phi : p\gamma \hookrightarrow p_1w_1 \in \Delta$ and $\text{st}(t_1) \dots \text{st}(t_n) \in \Phi$, then

$$\mathcal{C}[p(\gamma s, [t_1, \dots, t_n])] \rightarrow_M \mathcal{C}[p_1(w_1 s, [t_1, \dots, t_n])].$$

2. If $\Phi : p\gamma \hookrightarrow p_1w_1 \triangleright p_2w_2 \in \Delta$ and $\text{st}(t_1) \dots \text{st}(t_n) \in \Phi$, then

$$\mathcal{C}[p(\gamma s, [t_1, \dots, t_n])] \rightarrow_M \mathcal{C}[p_1(w_1 s, [p_2(w_2, []), t_1, \dots, t_n])].$$

Here, \mathcal{C} denotes a one-hole-context. We omit M , whenever it is clear from the context.

Given a set of configurations C , we define the set of backward reachable configurations $\text{pre}_M^*(C)$ by $\text{pre}_M^*(C) := \{t \mid \exists t_0 \in C : t \xrightarrow{*}_M t_0\}$. We want to compute $\text{pre}_M^*(C)$ for a given regular set of configurations C . Bouajjani et al. [3] showed that the set of backward reachable configurations is regular, if we start from a regular set of configurations and all constraints appearing in the CDPN are *stable*. A constraint Φ is called *stable* iff $\text{st}(t'_1) \dots \text{st}(t'_n) \in \Phi$, whenever $t_1, \dots, t_n, t'_1, \dots, t'_n$ are configurations such that $t_j \xrightarrow{*} t'_j$ holds for all $j = 1, \dots, n$, and $\text{st}(t_1) \dots \text{st}(t_n) \in \Phi$.

In order to do backward reachability analysis through linear Horn clauses, we first define a mapping between the least model of a set of Horn clauses with a special 4-ary predicate \mathbf{P} and a regular set of configurations. In the Horn clause framework, a control state p is a constant instead of a binary function symbol. For convenience, the states of the finite tree automaton for the set C , are represented by (finitely many) natural numbers.

For a set S of Horn clauses that contains a 4-ary predicate \mathbf{P} , and $i \in \mathbb{N}$, we define $\mathcal{L}_i(S)$ to be the set of configurations t such that the judgment $t \in \mathcal{L}_i(S)$ can be derived from derivations D for $\mathbf{P}(i, p, s, [i_1, \dots, i_n]) \in \mathcal{H}_S$ by the rule:

$$\frac{\begin{array}{c} \vdots D \\ \mathbf{P}(i, p, s, [i_1, \dots, i_n]) \end{array} \quad t_1 \in \mathcal{L}_{i_1}(S) \quad \dots \quad t_n \in \mathcal{L}_{i_n}(S)}{p(s, [t_1, \dots, t_n]) \in \mathcal{L}_i(S)}$$

Example 3. For the set $S = \{\mathbf{P}(1, p, \mathbf{0}, [2]) \Leftarrow \epsilon, \mathbf{P}(2, q, \mathbf{0}, L) \Leftarrow C(L), C(\square) \Leftarrow \epsilon, C([1]) \Leftarrow \epsilon\}$, we have $\mathcal{L}_1(S) = \{p(0, [q(0, \square)]), p(0, [q(0, [p(0, [q(0, \square)])])]), \dots\}$. \square

Finally we are able to describe an arbitrary regular set of configurations C by a set of Horn clauses this way:

Lemma 3. *Let $C \subseteq \text{Proc}$ be a regular set of configurations (given as a tree automaton). A set S_C of normal Horn clauses that contains a 4-ary predicate \mathbf{P} and a natural number $N \in \mathbb{N}$ can be constructed in linear time that fulfills the following properties:*

1. It holds $\mathcal{L}_1(S) \cup \dots \cup \mathcal{L}_N(S) = C$.
2. If $(i, p_1, s_1, l_1) \in \mathcal{H}_{S_C}(\mathbf{P})$ and $(i, p_2, s_2, l_2) \in \mathcal{H}_{S_C}(\mathbf{P})$, then $p_1 = p_2$.
3. If $(i, p_1, s_1, [i_1, \dots, i_n]) \in \mathcal{H}_{S_C}(\mathbf{P})$, then $\mathcal{L}_{i_j} \neq \emptyset$ for all $j = 1, \dots, n$. \square

Example 4. For $C = \{\#(\mathbf{0}, \square), \#(\mathbf{0}, [\#(\mathbf{0}, \square)]), \#(\mathbf{0}, [\#(\mathbf{0}, \square), \#(\mathbf{0}, \square)]), \dots\}$, the set

$$\begin{aligned} S_C = \{ & \mathbf{P}(x_N, x_P, x_S, x_L) \Leftarrow H_1(x_N) \wedge H_{\#}(x_P) \wedge H_{\mathbf{0}}(x_S) \wedge P_{list}(x_L), \\ & \mathbf{P}(x_N, x_P, x_S, x_L) \Leftarrow H_2(x_N) \wedge H_{\#}(x_P) \wedge H_{\mathbf{0}}(x_S) \wedge H_{\square}(x_L), \\ & \mathbf{P}_{list}(\square) \Leftarrow \epsilon, \mathbf{P}_{list}(x_N :: x_L) \Leftarrow H_2(x_N) \wedge P_{list}(x_L), H_{\mathbf{0}}(\mathbf{0}) \Leftarrow \epsilon, \\ & H_1(1) \Leftarrow \epsilon, H_2(2) \Leftarrow \epsilon, H_{\#}(\#) \Leftarrow \epsilon, H_{\square}(\square) \Leftarrow \epsilon \} \end{aligned}$$

is a set of normal Horn clauses that fulfills the properties mentioned in lemma [3](#). \square

The finite set of Horn clauses constructed according to lemma [3](#) for P represents the regular set C . We now add Horn clauses that describe backward reachability. Let $S_{CDPN}^1(M)$ be the smallest set of Horn clauses that fulfills the following properties:

1. If $\Phi : p\gamma \hookrightarrow p_1 w_1 \in \Delta$, then the following clauses are in $S_{CDPN}^1(M)$:

$$\mathbf{P}(x_N, p, \gamma x_S, x_L) \Leftarrow \mathbf{P}(x_N, p_1, w_1 x_S, x_L) \wedge Q_{\Phi}(x_L)$$

2. If $\Phi : p\gamma \hookrightarrow p_1 w_1 \triangleright p_2 w_2 \in \Delta$, then the following clauses are in $S_{CDPN}^1(M)$.

$$\mathbf{P}(x_N, p, \gamma x_S, x_L) \Leftarrow \mathbf{P}(x_N, p_1, w_1 x_S, x'_N :: x_L) \wedge \mathbf{P}(x'_N, p_2, w_2 \mathbf{0}, \square) \wedge Q_{\Phi}(x_L)$$

The predicate Q_{Φ} is a predicate that holds for a list $[i_1, \dots, i_n]$ iff there exists $t_j \in \mathcal{L}_{i_j}(S)$ for all $j = 1, \dots, n$ such that $\text{st}(t_1) \dots \text{st}(t_n) \in \Phi$. Consider for instance the constraint Φ , which is defined by the regular expression $(pq)^*$, where p and q are control states. The corresponding predicate Q_{Φ} can be defined by the following Horn clauses:

$$\begin{aligned} Q_{\Phi}(\square) & \Leftarrow \epsilon & Q_{\Phi}(x_N :: x_L) & \Leftarrow \mathbf{P}(x_N, p, x_1, x_2) \wedge Q_{\Phi'}(x_L) \\ Q_{\Phi'}(x_N :: x_L) & \Leftarrow \mathbf{P}(x_N, q, x_1, x_2) \wedge Q_{\Phi}(x_L) \end{aligned}$$

Systematically we do this as follows. We consider the constraint Φ to be given by the finite word automaton $\mathcal{A} = (Q, P, \delta, \Phi, F)$, where Q is the set of states, the set of control states P is the alphabet, $\delta \subseteq Q \times P \times Q$ is the transition relation, the constraint Φ is identified as the initial state, and F is the set of accepting states. The set $S_{\text{Constr}}(\mathcal{A})$ is the smallest set of Horn clauses that fulfills the following properties:

1. $Q_{\Phi'}(\Box) \Leftarrow \epsilon \in S_{\text{Constr}}(\mathcal{A})$, if $\Phi' \in F$.
2. $Q_{\Phi_1}(x_N :: x_L) \Leftarrow \mathbf{P}(x_N, p, x_1, x_2) \wedge Q_{\Phi_2}(x_L) \in S_{\text{Constr}}(\mathcal{A})$, if $(\Phi_1, p, \Phi_2) \in \delta$.

If we denote a finite word-automaton accepting Φ by \mathcal{A}_Φ , then the set $S_{\text{Constr}}(\mathcal{A}_\Phi)$ contains the Horn clauses that define the predicate Q_Φ . We collect all Horn clauses which are necessary to describe all constraints in the set $S_{\text{CDPN}}^2(M)$, i.e., $S_{\text{CDPN}}^2(M)$ is the union of all $S_{\text{Constr}}(\mathcal{A}_\Phi)$, where Φ is a constraint that appears in M . Finally we collect all Horn clauses from the sets $S_{\text{CDPN}}^1(M)$ and $S_{\text{CDPN}}^2(M)$ in the set $S_{\text{CDPN}}(M)$, i.e. we set $S_{\text{CDPN}}(M) := S_{\text{CDPN}}^1(M) \cup S_{\text{CDPN}}^2(M)$.

Example 5. For a CDPN M that consists of the transition rules $P^* : pa \hookrightarrow p \triangleright qa$, $P^* : qa \hookrightarrow \#$, and $\#^* : pa \hookrightarrow \#$, we get

$$\begin{aligned} S_{\text{CDPN}}^1(M) &= \{ \mathbf{P}(x_N, p, a x_S, x_L) \Leftarrow \mathbf{P}(x_N, p, x_S, x'_N :: x_L) \wedge \mathbf{P}(x'_N, q, a \mathbf{0}, \Box), \\ &\quad \mathbf{P}(x_N, q, a x_S, x_L) \Leftarrow \mathbf{P}(x_N, \#, x_S, x_L), \\ &\quad \mathbf{P}(x_N, p, a x_S, x_L) \Leftarrow \mathbf{P}(x_N, \#, x_S, x_L) \wedge Q_{\#^*}(x_L) \} \\ S_{\text{CDPN}}^2(M) &= \{ Q_{\#^*}(\Box) \Leftarrow \epsilon, Q_{\#^*}(x_N :: x_L) \Leftarrow \mathbf{P}(x_N, \#, x_1, x_2) \wedge Q_{\#^*}(x_L) \} \end{aligned}$$

For obvious reasons it is not necessary to define the predicate Q_{P^*} . □

For the remainder of the section let $C \subseteq \text{Proc}$ be a regular set of configurations and S_C be a set of normal Horn clauses that fulfills properties 2 and 3 of lemma 3 such that $\mathcal{L}_1(S_C) \cup \dots \cup \mathcal{L}_N(S_C) = C$ holds for some $N \in \mathbb{N}$. $S_C \cup S_{\text{CDPN}}(M)$ represents the pre_M^* -image of C precisely under the assumption that all constraints are stable:

Lemma 4. $\text{pre}_M^*(\mathcal{L}_i(S_C)) \subseteq \mathcal{L}_i(S_C \cup S_{\text{CDPN}}(M))$ for all i . Equality holds, whenever all constraints are stable. □

The set $S_C \cup S_{\text{CDPN}}(M)$ of Horn clauses only consists of linear Horn clauses. Actually, our normalization semi-procedure computes the least model of $S_C \cup S_{\text{CDPN}}(M)$ in at most exponentially many steps. We emphasize that the set $S_C \cup S_{\text{CDPN}}(M)$ of linear Horn clauses cannot be transformed into an equivalent finite set of \mathcal{H}_1 Horn clauses by instantiation. This is impossible, because the third argument and the fourth argument of the predicate \mathbf{P} are not always finite, i.e., the sets $\{t_3 \mid (t_1, t_2, t_3, t_4) \in \mathcal{H}_{S_C \cup S_{\text{CDPN}}(M)}(\mathbf{P})\}$ and $\{t_4 \mid (t_1, t_2, t_3, t_4) \in \mathcal{H}_{S_C \cup S_{\text{CDPN}}(M)}(\mathbf{P})\}$ can be infinite. The first argument of \mathbf{P} can be instantiated, since it is finite.

4 Conclusion

We extended the normalization procedure of Nielson et al. [12] for \mathcal{H}_1 Horn clauses to a normalization semi-procedure for linear Horn clauses. As a nontrivial application,

we demonstrated how to solve backward reachability for CDPNs by means of our normalization procedure. We have implemented our algorithm in OCAML. The system can be downloaded from <http://www2.in.tum.de/~seidl/downloads/H/>. Our system also provides support for the *Magic set* transformation which helps to narrow down the least model to those facts which are necessary to answer the query as well as special support for dealing with ground terms. Preliminary experiments with cryptographic protocols and dynamic pushdown networks seem promising. As future work, we want to try our tool on more realistic examples, and also enhance it with instantiation based on automatic finiteness analysis.

References

- [1] Blanchet, B.: An efficient cryptographic protocol verifier based on prolog rules. In: CSFW, pp. 82–96. IEEE Computer Society, Los Alamitos (2001)
- [2] Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997)
- [3] Bouajjani, A., Müller-Olm, M., Touili, T.: Regular symbolic analysis of dynamic networks of pushdown systems. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 473–487. Springer, Heidelberg (2005)
- [4] Dolev, D., Yao, A.C.-C.: On the security of public key protocols. IEEE Transactions on Information Theory 29(2), 198–207 (1983)
- [5] Finkel, A., Willems, B., Wolper, P.: A direct symbolic approach to model checking pushdown systems. Electr. Notes Theor. Comput. Sci. 9 (1997)
- [6] Goubault-Larrecq, J.: Personal communication (2003)
- [7] Goubault-Larrecq, J.: Deciding \mathcal{H}_1 by resolution. Inf. Process. Lett. 95(3), 401–408 (2005)
- [8] Goubault-Larrecq, J., Parrennes, F.: Cryptographic protocol analysis on real C code. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 363–379. Springer, Heidelberg (2005) ISBN 3-540-24297-X
- [9] Lugiez, D., Schnoebelen, P.: The regular viewpoint on pa-processes. Theor. Comput. Sci. 274(1-2), 89–115 (2002)
- [10] Mayr, R.: Decidability and Complexity of Model Checking Problems for Infinite-State Systems. PhD thesis, Technische Universität München (1998)
- [11] Nielsen, C.R., Nielson, F., Nielson, H.R.: Iterative Specialisation of Horn Clauses. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 131–145. Springer, Heidelberg (2008)
- [12] Nielson, F., Nielson, H.R., Seidl, H.: Normalizable Horn clauses, strongly recognizable relations and Spi. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 20–35. Springer, Heidelberg (2002)
- [13] Weidenbach, C.: Towards an automatic analysis of security protocols in first-order logic. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 314–328. Springer, Heidelberg (1999) ISBN 3-540-66222-7

A Graph-Based Implementation for Mechanized Refinement Calculus of OO Programs

Zhiming Liu¹, Charles Morisset², and Shuling Wang^{1,3}

¹ UNU-IIST, P.O. Box 3058, Macau S.A.R., China

² Royal Holloway, University of London
Information Security Group
Egham, Surrey TW20 0EX, U.K.

³ State Key Lab. of Computer Science
Institute of Software, Chinese Academy of Sciences

Abstract. This paper extends the mechanization of the refinement calculus done by von Wright in HOL, representing the state of a program as a graph instead of a tuple, in order to deal with object-orientation. The state graph structure is implemented in Isabelle, together with definitions and lemmas, to help the manipulation of states. We then show how proof obligations are automatically generated from the rCOS tool and can be loaded in Isabelle to be proved. We illustrate our approach by generating the proof obligations for a simple example, including object access and method invocation.

Keywords: Isabelle, Proof obligations, rCOS, Theorem proving.

1 Introduction

Software verification is about demonstrating that an *implementation* (executable code) of the software meets its *specification* (formal description of the behavior) and several techniques are available in order to achieve this goal. Testing and model checking usually aim to verify if a property holds on a subset of instances of a program, or on a model of the program, respectively, while theorem proving aims to build the proof of correctness, that is, the semantics of the implementation logically implies the specification. For all these techniques, there are several challenges to address:

i) An increasing number of software is written using the OO approach, and therefore the execution states of a program are complex, due to the complex relations among objects, aliasing, dynamic binding, and polymorphism. This makes it hard to understand and reason about the behavior of the program.

ii) When a tool is provided to help the development of software, it should offer an environment where the user can specify, analyze, implement and verify a program. Therefore, the different verification techniques need to be integrated within the tool.

iii) In general, it is not possible to automatically verify if an implementation satisfies a specification, therefore the tool is required to guide the user through the different steps of the verification process.

The refinement for Component and Object Systems (rCOS) [10,16] method provides an interesting framework to address these challenges. Firstly, rCOS has a formal semantics based on an extension of the Unifying Theories of Programming (UTP) [11] to include the concepts of components and objects. The graph-based operational semantics [12] has recently been defined for OO programs. Secondly, the rCOS tool (available at <http://rcos.iist.unu.edu>) provides a UML-like multi-view and multi-notational modeling and design platform. In particular, two verification processes are already implemented: the automated generation of test cases to check the robustness of a component [15], and the automated generation of CSP processes to verify the compatibility between the sequence diagram and the state diagram of a contract [7]. Lastly, rCOS extends the refinement calculus [11,17], which is a program construction method, where a non-deterministic specification is incrementally refined to deterministic code, using pre-defined rules. This approach creates several refinement steps, which fill the gap between the specification and the implementation, therefore reduces the proof complexity, by replacing a single complex proof by many simpler ones.

Related Work. The mechanization of the refinement calculus was firstly done in [24], which has been extended to include pointers [2] and also object-oriented programs [4,20]. In particular, a refinement calculus has been defined for Eiffel contracts [19], and encoded in PVS [18]. Although this approach addresses a similar issue than the one exposed here, the authors encode the calculus using a shallow embedding, that is, a class in Eiffel is encoded as a type in PVS, a routine in Eiffel is encoded as a function in PVS, etc. Proofs of refinement are then done over PVS *programs* rather than PVS *terms*, and so require the understanding of the underlying semantics of PVS. We use here a deep embedding, following [24], and the proofs of refinement are done, roughly speaking, over the abstract syntax tree of the original program, and so only require to know how to write a proof in Isabelle/Isar. The Program Refinement Tool [3] provides a deep embedding of a refinement calculus, and even if it does not support OO programs natively, it could be extended with an existing formalization which does [22]. However, rCOS also provides a semantics for components, and even if we do not address in this paper the issue of verification of component protocols, this work is part of a larger framework where other verification techniques exist [21]. In other words, the work presented here is not a standalone tool, but adds up to a collection of tools that helps a developer to specify, implement and verify an application.

On the other hand, different memory models for object-oriented programs have been encoded in theorem provers [9,23,13]. However, the memory in these approaches is either modelled as a function from addresses or pointers to values or using records to represent objects. Although such a modelling is very expressive, and has been shown to be adapted to automated demonstration, we propose here a representation of the memory by a directed and labeled graph, that is intuitive than a representation by a function or set of records. The graph structure helps in the formulation of properties and carrying out interactive proofs.

Contribution. The main contribution of this paper is twofold. The first one is the implementation in Isabelle of rCOS using a graph to represent the state of a program. It is an extension of the mechanization of the refinement calculus done by von Wright in HOL [24]. The other one is the automated generation of proof obligations for refinement steps in rCOS. Concretely, we have implemented in the rCOS tool a plug-in which is able to take the bodies of two methods defined with the rCOS language, translate each body into a predicate transformer in Isabelle and generate automatically an Isabelle lemma stating that the first predicate transformer refines the second one (the proof still has to be done by the user). This process is linked with the definition of refinement steps within the tool. The most technical part of this work is the translation from rCOS designs to Isabelle statements.

Organization. Section 2 introduces the rCOS language. Section 3 recalls the previous mechanization of the refinement calculus. Section 4 presents the graph-based representation of the memory and its implementation in Isabelle/HOL. Section 5 extends the mechanization of refinement calculus to object-orientation. Section 6 presents an example to illustrate our approach. Finally, Section 7 concludes and presents the future work.

2 rCOS

The rCOS method consists of two parts: a component/object-oriented language, with a formal semantics, and a modeling tool, enforcing a use-case based methodology for software development, providing tool support and static analysis. We give here a brief description of the language, and we refer to [5,6] for further details.

2.1 Language

The rCOS language is an extension of UTP [11], to include object-oriented and component features. The essential theme of UTP that helps rCOS is that the semantics of a program P (or a statement) in any programming language can be defined as a predicate, called a *design*. The most general form of a *design* is a pair of pre- and post-conditions [11,17], denoted as $p(x) \vdash R(x, x')$, of the *observable* x of the program. Its meaning is defined by the predicate $p(x) \wedge \text{ok} \Rightarrow R(x, x') \wedge \text{ok}'$, which asserts that if the program executes from a state where the *initial value* x satisfies $p(x)$, the program will terminate in a state where the final value x' satisfies the relation $R(x, x')$ with the initial value x . Observables include program variables and parameters of methods or procedures. The Boolean variables ok and ok' represent observations of termination of the execution preceding the execution of P (*i.e.* ok is true) and the termination of the execution of P (*i.e.* ok' is true), respectively. Non-deterministic choice is defined as $d_1 \sqcap d_2$, where d_1 and d_2 are designs.

The language also includes traditional imperative statements, and a design can be: SKIP and CHAOS, an assignment $p := e$, where p is a navigation path,

and e is an expression; a conditional statement $d_1 \triangleleft b \triangleright d_2$, where d_1 and d_2 are designs and b is a boolean expression; a sequence $d_1; d_2$, where d_1 and d_2 are designs; a loop **do** b d , where d is a design and b is a boolean expression; a local variable declaration and un-declaration **var** T $x = e$; **end** x , where T is a type.

Objects are created through the command $C.\text{new}(p)$, where C is a class type and p is a navigation path. It creates a new object of type C whose attributes have the initial values as declared in C , and attaches the new object to p . A method invocation has the form $e.m(\text{ve}, \text{re})$, where m is a method and e , ve , re are expressions. Intuitively, it first records the value of the actual value parameter ve in the formal value parameter of m , and then executes the body of m . At the end it returns the value of the formal return parameter to the actual return parameter re .

The rCOS language includes the notion of components, which provide or require contracts. A contract includes an interface (a set of field and method declarations), the specification of each method and a protocol stating the allowed sequences of method calls (for instance, for a buffer, the method `put` must be called before the method `get`). A component provides a contract through a class, which is the usual notion of class, where each method has to be defined using a design. Note that the design of a method does not have to be executable in general, only if the user wants to generate Java code, since executable rCOS designs are quite similar to Java programs. For instance, all the following examples are correct rCOS programs.

```

class A {
  int x;
  public m(int v) { x := v }
}

class B1 {
  A a;
  public foo() {
    [ true |− a.x' = 2 ∨ a.x' = 3 ] }
}

class B2 {
  A a;
  public foo() {
    [ true |− a.x' = 1 ] ;
    a.x := a.x + 1 }
}

class B3 {
  A a;
  public foo() {
    a.m(1) ; a.x := a.x + 1 }
}

```

The method $B_1::\text{foo}$ is abstract and non-deterministic: it just specifies, under the true precondition, that the value of the field x of the field a should be either equal to 2 or to 3. The method $B_2::\text{foo}$ mixes abstract pre/post-conditions with a concrete assignment while $B_3::\text{foo}$ is completely concrete and could be directly translated to Java. In this example, we can see that $B_1::\text{foo}$ is refined by $B_2::\text{foo}$, which is refined by $B_3::\text{foo}$. We detail in the following section the mechanization of the notion of refinement.

3 Mechanized Refinement

The refinement calculus [117] is a program construction method, where a non-deterministic specification is incrementally refined to deterministic code, using

pre-defined rules. This calculus has been fully implemented with a theorem prover, HOL, in [24,8] and then extended, in particular in [14], which introduces, among others, procedures and recursive functions. The implementation is actually the definition of a predicate transformer semantics, i.e. the weakest precondition. For any design d and any predicate q over states, the function $d\ q$ is defined as the weakest precondition that should be true on states before executing d such that q holds after executing d . Therefore, a design is usually considered as a predicate transformer, since it takes a predicate (q) as input and returns another predicate (the weakest precondition of q). We recall here the definitions of assignment and refinement from [24]. We use `State` to represent the type of program states. We introduce first the types of predicates over states and the type of predicate transformers.

types `State pred = State \Rightarrow bool`
`State predT = State pred \Rightarrow State pred`

The `assign` predicate transformer takes a function `e`, which takes a state and returns the state where the corresponding assignment is done. The weakest precondition of a predicate `q` is calculated by checking `q` on a state where the assignment has been done.

definition `assign :: (State \Rightarrow State) \Rightarrow (State predT) where`
`assign e q \equiv λ u. q (e u)`

A design `c1` is refined by a design `c2` if, and only if, the weakest precondition of `c1` implies the one of `c2` for any state.

definition `implies :: (State pred) \Rightarrow (State pred) \Rightarrow bool where`
`implies p q \equiv \forall u. (p u) \Rightarrow (q u)`

definition `refines :: (State predT) \Rightarrow (State predT) \Rightarrow bool (infixl ref 40) where`
`c1 ref c2 \equiv \forall q. (implies (c1 q) (c2 q))`

In addition to the definition of the semantics, helpful theorems are introduced in [24]. For instance the one stating that the loop `do g c` refines the loop `do g d` if the design `c` refines the design `d`.

theorem `do_ref : d ref c \Rightarrow (do g d) ref (do g c)`

Although the previous definitions do not directly depend on the structure of the state, the latter is defined as a tuple [24], where each element of the tuple is the value of a variable of the program. For instance, if a program has two variables x and y , set respectively to 1 and 3, the state of such a program is the pair $(1, 3)$. The names of the variables are therefore lost in the translation, and any operation concerning x has to be translated as an operation concerning the first element of the pair. As a result, dealing with local variables and method calls implies to extend and narrow the state, respectively. Moreover, this approach does not directly handle references and therefore such a representation for states cannot be applied for OO programs. The usual way to tackle this issue is to represent OO states as records or as a function from pointers to values [2,19,4,20]. A new approach uses graphs instead [12], and we present it in the next section.

4 Graph Representation

In [12], the state of a program is represented as a directed labeled graph. We recall here the definition of such a graph and give its implementation in Isabelle/HOL, together with basic operations to manipulate state graphs.

4.1 State Graph

A state graph describes the values of variables, together with a family of objects and their relations. Due to the existence of nested local variables and method invocations, we also need to describe scopes in state graphs. A scope is represented as a node in a state graph, called *scope node*. Two scope nodes are adjacent iff the scopes they represent are directly nested. They are connected by an edge labeled by \$, the one corresponding to the inner scope as the source and the other one as the target of the edge respectively. In particular, the top scope node with no incoming \$ edge represents the current scope, and is thus the current root of the state graph. For instance, in Fig. 1(a), r is the root of the graph.

The outgoing edges of a scope node, except for the \$ edge, represent the variables defined in the corresponding scope. A non-scope node in a state graph represents an object or a primitive datum, called *object node* and *value node*, respectively. An object node is labeled by the runtime type of the object, while a value node is labeled by the primitive value. An outgoing edge from an object node is labeled by a field name of the source object and refers to the target object representing the value of this field. There is no outgoing edge from a value node.

Let \mathcal{A} be the set of names of variables (including the special variable *this* which refers to the current object) and fields, and $\mathcal{A}^+ = \mathcal{A} \cup \{\$\}$. Let \mathcal{C} be the set of classes and \mathcal{W} the set of constant values. The formal definition of a *state graph* is then given as follows.

Definition 1 (State Graph). *A state graph is a rooted, directed and labeled graph $G = \langle V, E, T, F, r \rangle$, where*

- $V = R \cup N \cup L$ is the set of nodes, where R is the set of scope nodes, N is the set of object nodes and L is the set of value nodes,
- $E \subseteq V \times \mathcal{A}^+ \times V$ is the set of edges,
- $T : N \rightarrow \mathcal{C}$ is a mapping from object nodes to types,
- $F : L \rightarrow \mathcal{W}$ is a mapping from value nodes to values,
- $r \in R$ is the root of the graph and it has no incoming edges,
- starting from r , the \$-edges, if there are any, form a path such that except for r each node on the path has only one incoming edge.

All the nodes on the \$-path are scope nodes, the top of which is the root of the state graph. When a new scope is entered, a new node together with a \$-edge from it to the current root are pushed onto the \$-path; and when a scope is exited, the top node of the \$-path is popped out, together with all edges outgoing from it. As an illustration, Figure 1(a) shows the state graph after the command `a.b.x :=1; var int c=2;` is executed. Moreover, Figure 1(b) shows a state graph with recursive objects, where the types and values of nodes are ignored.

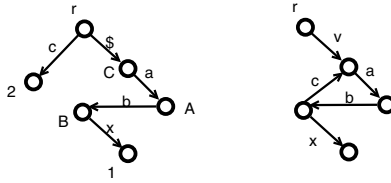


Fig. 1. (a): a state graph; (b): a state graph with recursive objects

4.2 Graph Implementation

A state graph requires the sets of scope nodes, object nodes and value nodes to be disjoint. We therefore define the datatype **vertex** as the union of four disjoint types: **onode** for object nodes, **snode** for scope nodes, **vnode** for value nodes, and \perp for the undefined vertex, the latter being introduced mainly for the definition of graph operations.

datatype vertex = N onode | R snode | L vnode | \perp

A state graph is defined as the cartesian product of four elements:

types

edgefun = vertex \Rightarrow label \Rightarrow vertex
 onodefun = onode \Rightarrow ctype
 vnodefun = vnode \Rightarrow val
 graph = edgefun * onodefun * vnodefun * snode list

The first element of a graph is the function **edgefun**, which given a vertex a and a label x , returns the vertex b if (a, x, b) is in the set of edges, \perp otherwise (note that \perp is different from the node corresponding to *null*). Such a definition automatically ensures the determinism for edges. The second (resp. the third) element of a graph corresponds to the function T (resp. the function F). The last element is a list of scope nodes, where the head of the list stands for the current root, the second element stands for the previous one, and so on. This representation of scope nodes allows us not to implement $\$$ -edges.

In order to ensure that there is no edge starting with the undefined vertex \perp , we introduce the following property.

definition isGoodFunction:: graph \Rightarrow bool **where**
 isGoodFunction g $\equiv \forall x. (\text{getEdgeFun } g) \perp x = \perp$

where **getEdgeFun** g is used to get the first component of g.

Moreover, we need some properties concerning the list of scope nodes. Indeed the **snode** list is well-formed (and in this case the graph satisfies **isCorrectSnode**, which we do not detail here due to lack of space) if, and only if: (1) the scope nodes cannot be undefined; (2) all the scope nodes are unique; (3) from each scope node, there must exist at least one outgoing edge; (4) a scope node can never be the target of an edge. Thus, a graph **g** is *well-formed*, denoted by **wfGraph** g if, and only if, it satisfies **isGoodFunction** and **isCorrectSnode**.

4.3 Graph Operations

This section gives the implementation of some basic graph operations. Due to space limitation we only present a subset of definitions below, more can be found at http://isg.rhul.ac.uk/morriset/sbmf/graph_utl.thy.

First, the function `swingEdge` swings an edge with the given source and label to point to a new vertex by updating the `edgefun` of the graph. It will return the original graph when the edge to be swung does not exist, or when the new target is undefined.

definition `swingEdge`:: vertex \Rightarrow label \Rightarrow vertex \Rightarrow graph \Rightarrow graph **where**
`swingEdge n x m g` \equiv (**if** ((`getEdgeFun g`) n x) = \perp **then** g
else if m = \perp **then** g
else let new_EF = ((λ v. λ l. (**if** (v = n & l = x) **then** m
else ((`getEdgeFun g`) v l))) **in** (new_EF, snd g))

We then introduce the type `path` as a list of labels, hence representing navigation paths. For implementation optimisation reasons, the path is actually stored as the reverse list of labels: for instance, the rCOS expression `a.b.x` is defined as the list [`'x'`, `'b'`, `'a'`]. Given a path `p` and a graph `g`, the function `getOwner` returns the vertex that the tail of the path `p` refers to in the graph. Formally, it returns \perp if the path is empty, the corresponding scope node of the variable if the path is limited to one element (using the function `getSnodeOfVar`) and otherwise, recursively gets the owner of the tail of the path, from which it gets the next vertex associated with the head of the tail.

consts `getOwner` :: path \Rightarrow graph \Rightarrow vertex
primrec `getOwner` [] g = \perp
`getOwner (x#t) g` = (**if** t = [] **then** (R (`getSnodeOfVar x g`))
else ((`getEdgeFun g`) (`getOwner t g`) (hd t)))

Starting from the source `getOwner p g` and the label `hd p`, we can reach the target vertex that the path `p` refers to in `g`, which is exactly the definition of the function `getVertexPath`. This operation corresponds to the general evaluation of path expressions in a program state.

definition `getVertexPath` :: path \Rightarrow graph \Rightarrow vertex **where**
`getVertexPath p g` \equiv (**if** p = [] **then** \perp
else ((`getEdgeFun g`) (`getOwner p g`) (hd p)))

Finally, the function `swingPath` swings the last edge of a path in the graph to point to a new vertex. In other words, it sets a new value to a path in a state graph, and therefore, can be used for implementing assignments in rCOS. As defined below, it uses `getOwner` and `hd p` to find the source and the label of the last edge respectively, and then swings the edge by using `swingEdge` directly. When the path is empty, `swingPath` returns the original graph.

definition `swingPath` :: path \Rightarrow vertex \Rightarrow graph \Rightarrow graph **where**
`swingPath p n g` \equiv (**if** p = [] **then** g **else** `swingEdge` (`getOwner p g`) (hd p) n g)

This function has been proved to preserve the well-formedness, *i.e.* for any graph `g`, any path `p` and any non-scope node vertex `n`, if `g` is well-formed then (`swingPath p n g`) is also well-formed.

Furthermore, we prove that a path after being swung to a vertex will actually point to the vertex. Before stating the fact, we define a path p to be well-formed w.r.t. g , denoted by $\text{wfPath } p \ g$, if, and only if, the vertex of p exists in g , and the owner of p appears exactly once as a source in the list of edges along the path, the latest of which is exactly the property $\text{isGoodPath } p \ g$. For instances, in the state graph in Figure 11(b), the paths $v.a$, $v.a.b.c$ and $v.a.b.x$ satisfy isGoodPath , while $v.a.b.c.a$ and $v.a.b.c.a.b.x$ do not. We discuss in the conclusion about the limitations caused by this constraint.

Finally, the theorem $\text{swingPathChangeVertex}$ is proved under three assumptions: g is well-formed; p is well-formed w.r.t. g ; and n is not the undefined vertex. In particular, with the assumption isGoodPath , we can prove a key fact that the owner of the path p is not changed after it is swung.

theorem $\text{swingPathChangeVertex}$:

$\text{wfGraph } g \Rightarrow \text{wfPath } p \ g \Rightarrow n \neq \perp \Rightarrow \text{getVertexPath } p \ (\text{swingPath } p \ n \ g) = n$

We will introduce functions for implementing local variable (un-)declaration. Adding edges representing variables in a graph is done by the function addVarList , which given a function f of type labelVerF (mapping variables to their initial values) and a graph g , adds edges labeled with variables lp in the domain of f to the current root of g and lets them point to the associated vertices $f \ lp$.

definition $\text{addVarList} :: \text{labelVerF} \Rightarrow \text{graph} \Rightarrow \text{graph}$ **where**

$\text{addVarList } f \ g \equiv \text{case } (\text{getSnodeList } g) \text{ of } [] \Rightarrow g$
 $| a\#p \Rightarrow ((\lambda \text{vp}. \lambda \text{lp}. (\text{if } (\text{vp} = (\text{R } a) \ \& \ (\exists v. v \neq \perp \ \& \ f \ lp = v)) \text{ then } (f \ lp)$
 $\text{else } ((\text{getEdgeFun } g) \ \text{vp} \ \text{lp}))), \text{snd } g)$

Moreover, we define a function createSnode which pushes a new scope node into the scope node list of a graph. The function vars then combines the operations of creating a scope node and adding edges, and therefore implements local variable declaration.

definition $\text{Vars} :: \text{labelExpF} \Rightarrow \text{graph} \Rightarrow \text{graph}$ **where**

$\text{Vars } f \ g \equiv \text{addVarList } (\text{expToNode } f \ (\text{createSnode } g)) \ (\text{createSnode } g)$

where labelExpF maps variables to their initial expressions, and expToNode translates a function of type labelExpF to the corresponding one of type labelVerF , by changing in the range the expressions to their values in the graph. Finally, we introduce a function removeSnode which removes the top root from the scope node list, and in consequence all edges outgoing from the scope node. It implements local variable un-declaration.

definition $\text{removeSnode} :: \text{graph} \Rightarrow \text{graph}$ **where**

$\text{removeSnode } g \equiv \text{case } (\text{getSnodeList } g) \text{ of } [] \Rightarrow g$
 $| t\#q \Rightarrow (\lambda v. \lambda l. (\text{if } (v = \text{R } t) \text{ then } \perp \text{ else } (\text{getEdgeFun } g) \ v \ l),$
 $\text{getOnodeFun } g, \text{getVnodeFun } g, q)$

where the functions getOnodeFun and getVnodeFun are used to get the second and third components of a graph respectively.

We are now in position to introduce the function `addObject` which creates a new vertex (object) in a graph. Given a path `l`, a class type `s` and a graph `g`, this function first gets a fresh object node of type `s` not in `g`, done by `getNodeFromType`; then swings the path `l` to refer to the new vertex by using `swingPath`; and finally, attaches the attributes of class `s` to the new node and initialises them, which is implemented by the function `addAttrs`. In the definition, the function `getAttrsOfCtype` is used to extract the attribute information from class `s`.

definition `addObject` :: path \Rightarrow ctype \Rightarrow graph \Rightarrow graph **where**
`addObject l s g` \equiv **let** `v` = (N (getNodeFromType s g)) **in** `addAttrs v`
 (expToNode (getAttrsOfCtype s) g) (swingPath l v g)

The functions `removeSnode`, `Vars` and `addObject` are proved to preserve the graph well-formedness, and furthermore, the last two are proved to ensure that variables or attributes are initialised with the correct values.

5 Refinement of rCOS Designs

The graph-based representation of the memory presented in the previous section allows us to extend the mechanization of the refinement calculus presented in Section 3 to deal with object-orientation. Since we only consider well-formed graphs and paths, we integrate these conditions into the weakest precondition of each command. The complete definition of the refinement calculus for all constructs can be found at <http://isg.rhul.ac.uk/morriset/sbmf/rcos.thy>.

5.1 Primitive Designs

Pre/post-condition. The definition of the non-deterministic assignment is changed to include the well-formedness checks.

definition
`nondass` :: (graph \Rightarrow graph pred) \Rightarrow path list \Rightarrow (graph pred) \Rightarrow (graph pred) **where**
`nondass P l q` \equiv (λv . (wfGraph v) & (wfPathl l v) & ($\forall v1$. P v v1 \Rightarrow q v1))

where `wfPathl l v` is true if, and only if, every path in `l` satisfies `wfPath`. This list of paths corresponds to all the paths appearing in the post-condition. A pre/post-condition is then an assertion followed by a non-deterministic assignment.

definition `pp` :: (graph pred) \Rightarrow (graph \Rightarrow graph pred) \Rightarrow path list \Rightarrow
 (graph predT) **where**
`pp p r l` \equiv `assert p ; nondass r l`

where `assert` is the standard definition for the assertion. For instance, the pre/post-condition [`true` |− a.b.x'=2] is translated into the following statement

`pp (true) (λg . $\lambda g1$. ((getIntOfPath a.b.x g1) = 2)) [a.b.x]`

where, for the sake of readability, we write a path as in code, e.g. *a.b.x* stands for the path `[''x'', ''b'', ''a'']`.

Assignment. The definition of the assignment is changed as follows.

definition `assign` :: `path` \Rightarrow `exp` \Rightarrow (`graph pred`) \Rightarrow (`graph pred`) **where**
`assign p e q` \equiv λu . `wfGraph u` & `wfPath p u` & `wfExp e u` &
`q (swingPath p (getNodeExp e u) u)`

where the path `p` is assigned to the expression `e`, which is required to be well-formed. The function `getNodeExp` returns the value of an expression, which is defined using `getVertexPath` when the expression to be evaluated is a path, otherwise itself when it is a constant value.

Local Declaration and Un-declaration. The commands `begin` and `end` declare/initialize new local variables and terminate them, respectively.

definition `begin` :: `labelExpF` \Rightarrow (`graph pred`) \Rightarrow (`graph pred`) **where**
`begin f q` \equiv λu . `wfGraph u` & `wfLabelExpF f u` & `q (Vars f u)`

definition `end` :: (`graph pred`) \Rightarrow (`graph pred`) **where**
`end q` \equiv λu . `wfGraph u` & `q (removeSnode u)`

where `f` is a well-formed function of type `labelExpF`, which means that for each local variable, it is initialised by a well-formed expression in `f`.

The command `locdec` defines the block for local declaration and un-declaration, where `f` is the same as above and `c` is the body of the block.

definition `locdec` :: `labelExpF` \Rightarrow (`graph predT`) \Rightarrow (`graph predT`) **where**
`locdec f c` \equiv `begin f`; `c`; `end`

Method Invocation. The command `method` implements a method invocation with the help of the command `locdec`.

definition `method` :: (`label * exp`) `list` \Rightarrow (`graph predT`) \Rightarrow (`graph predT`) **where**
`method l c` \equiv `locdec (getLabelExpF l) c`

where `l` is of type (`label * exp`) `list`, each pair consisting of a formal value parameter and the corresponding actual value parameter of the method, and `c` is the method body followed by the assignment from the formal return parameter to the actual return parameter. In the `method` command, the function `getLabelExpF` translates a list of pairs of type `label * exp` to the corresponding mapping of type `labelExpF` (i.e. `label` \Rightarrow `exp`). For instance, the method call `a.m(1)` in the example of Section 2 is translated as:

```
method [(this, Path this.a), (v, Val (Zint 1))] ;assign this.x Path v
```

When the method is called, the variable `this` is initialised by `this.a` (the caller), and `v` by 1. Note that with this approach, recursive method calls are not directly handled, and require the definition of a fix-point, which we do not consider here.

lemma `ref_pp_assign` :

```
pp (true) (λ g. λ g1. ((getNatOfPath g1 p) = n)) [p] ref
(assign p (Val (Zint n)))
```

The proof of this lemma, together with the proofs of other useful lemmas, can be found at http://isg.rhul.ac.uk/morisset/sbmf/rcos_lib.thy.

Another example is the Expert Pattern, which is an essential rule for OO functionality decomposition by delegating responsibilities through method calls to the objects, called the experts, that have the information to carry out the responsibilities. For instance, defining a setter for a field is a special case of the Expert Pattern, and therefore a refinement. In this case, we have proved, with the theorem `EPIsRefOne`, that the method `bar() { p.x := n }` is refined by the method `bar() { p.m() }` where `m () {this.x := n}` is a method of `p`, for any non-empty path `p` and constant `n`; a similar theorem `EPIsRefTwo` is provided, when the setter takes the value as an argument, that is, the method `bar() { p.x := n }` is refined by the method `bar() { p.m(n) }` where `m (T v) {this.x := v}` is a method of `p`, for any primitive type `T` and parameter `v`.

6.3 Example

Let us consider the examples given in Section 2. The user first defines the classes A and B_1 and then introduces a manual refinement, concerning the operation `foo`, where the old design is $d_{old} = [\mathbf{true} | -a.x'=2 \vee a.x'=3]$ and the new design is $d_{new} = \{a.m(1) ; a.x := a.x + 1\}$. When applying the refinement, the class B_3 is obtained. However, proving directly that d_{old} is refined by d_{new} might be complex, as several steps of refinement are involved. The user can either decompose the refinement proof in Isabelle or directly in the rCOS tool. Indeed, the latter allows one to easily compose refinement steps. In this example, the user can introduce the manual refinement r_1 of $[\mathbf{true} | -a.x'=2 \vee a.x'=3]$ to $[\mathbf{true} | -a.x'=2]$; the automatic refinement r_2 of pre/post-conditions to assignments; the the manual refinement r_3 of $a.x := 2$ to $a.x:=1$; $a.x := a.x+1$; the expert-pattern refinement r_4 on $a.x := 1$.

The proof obligations for each step can be generated automatically. For instance, the proof obligation corresponding to r_3 is the following statement:

```
assign this.a.x (Val (Zint 2)) ref
(assign this.a.x (Val (Zint 1)));
(assign this.a.x (Plus (Path this.a.x) (Val (Zint 1))))
```

The refinement r_1 strengthens the post-condition of the design, so the proof is quite straight-forward. The proofs of r_2, r_3 and r_4 directly follow from the lemmas described in the previous subsection. Finally, using the fact that the refinement relation is transitive, we can prove that $d_{old} = [\mathbf{true} | -a.x'=2 \vee a.x'=3]$ is refined by $d_{new} = \{a.m(1) ; a.x := a.x + 1\}$. The complete proofs can be found at <http://isg.rhul.ac.uk/morisset/sbmf/example.thy>. Except the proofs of r_1 and r_3 , all the other proofs could be derived automatically, since automatic

transformations are used, meaning we can directly use the lemmas associated with these transformations. Such an automatic association is, as we say in the conclusion, a future work.

7 Conclusion

The approach presented in this paper allows a user of the rCOS tool to automatically generate proof obligations of design refinement. The generated statements are defined using the predicate transformer semantics mechanized in [24] extended here to support object-oriented programs, thanks to a graph-based representation of the memory. A library of lemmas and theorems is available to the user in order to help her to prove the generated statements, concerning for instance the graph operations or the refinement calculus. There are two major strengths to this approach. The first one is the seamless integration within the rCOS tool, hence making the process transparent for the user, who can design a software using UML, with the proof obligations being automatically generated. Of course, these obligations still have to be discharged, but using a more generic back-end, like Why [9], would generate proof statements both for interactive theorem provers and automated demonstrators. However, since we are using higher-order terms, automated demonstration might not be very efficient, therefore we need to keep providing lemmas corresponding to refinement steps, especially the ones concerning OO concepts. This issue poses the one of the scalability, since large programs usually involve a large number of refinement rules and a statement containing a large number of rules can only be proven in a reasonable amount of time and space if each rule has been proven for the general case. The proof would then only be a succession of application of simple rules, which is the main strength of the refinement calculus. We therefore believe that the effort should be made to provide the developer with a large number of rules already proven rather than trying to automatically prove a complex transformation.

The other strength of this approach is the definition of the graph-based semantics. Although using a graph is not strictly more expressive than using records or a function from pointers to values, that is, it does not allow one to express more programs, the properties of a state can be expressed in a different, more abstract and intuitive way. In particular, the graph model helps the formulation and understanding of properties in the first place and on the other hand provides intuition for construction of their proofs to be checked by the theorem prover. In practice, the function `getEdgeFun` of a graph is conceptually close to a function from pointers to values: each object node is a pointer and each value node is a value. In other words, a graph can be seen as an extensional definition of a usual function from pointers to value. However we believe that a more abstract view of the memory helps reasoning about the state of the program. For instance, stating that a path `p` is alias-free in a state `g` is simply done by stating that there is no path `p2` different from `p` such that `getVertexPath p g = getVertexPath p2 g`.

Several limitations need however to be addressed, for instance the assumption `isGoodPath` in the theorem `swingPathChangeVertex` or in the weakest precondition

of the statements, which may be too strong. Intuitively, this theorem still holds without it but the proof is more complex, since in general we lose the fact that the owner of the path is the same before and after swinging, as we showed on the examples. A possible lead to address this issue is to consider that any path which does not satisfy `isGoodPath` can be “reduced” to a path which does. For instance, on Figure 1(b), the path `v.a.b.c.a` points to the same object that `v.a` points to, but `v.a` satisfies `isGoodPath`.

Moreover, more designs need to be implemented in the translation process, for instance recursive method calls. However, we can extensively reuse [8,14], defined for imperative programs, by extending them to object-oriented programs. The method call does not currently support dynamic binding, but it could be done by looking up the actual type of the caller from the second element `onodefun` of the graph to fix the called method body.

In general, the next step is to integrate this mechanism within model transformations, which is an on-going work in the rCOS tool [21]. The principal challenge in this work is for the tool to handle several models at the same time: before, during and after refinement. Such modifications are easy to handle when changing the design of a single method, but more complicated when for instance changing or deleting classes. The idea is then to establish a correspondence between the modifications done in the tool and the refinement rules involved. Moreover, a better interaction with Isabelle could help the user in the decomposition of the refinement steps. For instance, by using similar techniques than those used in automated demonstration, the tool could use the feedback from the theorem prover to ask the user to introduce more steps. We then could have a system where the user introduces a refinement rule, the tool tries to prove it automatically, if it cannot, it asks the user for an intermediary step, tries again, and so on until the whole rule is proved.

Acknowledgment. This work has been supported by the project GAVES of the Macao S&TD Fund, the 973 program 2009CB320702, STCSM 08510700300, and the projects NSFC-60721061, NSFC-60970031, NSFC-90718041 and NSFC-60736017. The authors would like to thank Volker Stolz for his useful remarks.

References

1. Back, R.-J.: On the Correctness of Refinement Steps in Program Development. PhD thesis, Helsinki, Finland, Report A-1978-4 (1978)
2. Back, R.-J., Fan, X., Preoteasa, V.: Reasoning about pointers in refinement calculus. Technical Report 543, TUCS - Turku Centre for Computer Science, Turku, Finland (July 2003)
3. Carrington, D., Hayes, I., Nickson, R., Watson, G., Welsh, J.: A tool for developing correct programs by refinement. In: Proc. BCS 7th Refinement Workshop. Springer, Heidelberg (1996)
4. Cavalcanti, A., Naumann, D.A.: A weakest precondition semantics for refinement of object-oriented programs. IEEE Transactions on Software Engineering 26, 713–728 (2000)

5. Chen, Z., Liu, Z., Ravn, A.P., Stolz, V., Zhan, N.: Refinement and verification in component-based model driven design. *Science of Computer Programming* 74(4), 168–196 (2009); UNU-IIST TR 388
6. Chen, Z., Liu, Z., Stolz, V.: The rCOS tool. In: *Modelling and Analysis in VDM: Proceedings of the Fourth VDM/Overture Workshop*, number CS-TR-1099 in Technical Report Series. Newcastle University (May 2008)
7. Chen, Z., Morisset, C., Stolz, V.: Specification and validation of behavioural protocols in the rCOS modeler. In: Arbab, F., Sirjani, M. (eds.) *FSEN 2009*. LNCS, vol. 5961, pp. 387–401. Springer, Heidelberg (2010)
8. Depasse, C.: *Constructing Isabelle proofs in a proof refinement calculus*. Research Report, UCL (2001)
9. Filliâtre, J.-C.: *Why: a multi-language multi-prover verification tool*. Research Report 1366, LRI, Université Paris Sud (2003)
10. He, J., Liu, Z., Li, X.: rCOS: A refinement calculus of object systems. *Theor. Comput. Sci.* 365(1-2), 109–142 (2006); UNU-IIST TR 322
11. Hoare, C.A.R., He, J.: *Unifying Theories of Programming*. Prentice-Hall, Englewood Cliffs (1998)
12. Ke, W., Liu, Z., Wang, S., Zhao, L.: A graph-based operational semantics of oo programs. In: Breitman, K., Cavalcanti, A. (eds.) *ICFEM 2009*. LNCS, vol. 5885, pp. 347–366. Springer, Heidelberg (2009)
13. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.* 28(4), 619–695 (2006)
14. Laibinis, L.: *Mechanised Formal Reasoning About Modular Programs*. PhD thesis, Abo Akademi (2000)
15. Lei, B., Liu, Z., Morisset, C., Li, X.: State based robustness testing for components. In: *FACS 2008*. ENTCS, vol. 260, pp. 173–188. Elsevier, Amsterdam (2008)
16. Liu, Z., Morisset, C., Stolz, V.: rCOS: theory and tools for component-based model driven development. In: Arbab, F., Sirjani, M. (eds.) *FSEN 2009*. LNCS, vol. 5961, pp. 62–80. Springer, Heidelberg (2010)
17. Morgan, C.: *Programming from specifications*, 2nd edn. Prentice Hall International, Englewood Cliffs (1994)
18. Paige, R., Ostroff, J., Brooke, P.: Formalising eiffel references and expanded types in pvs. In: *Proc. International Workshop on Aliasing, Confinement, and Ownership in Object-Oriented Programming* (2003)
19. Paige, R.F., Ostroff, J.S.: ERC – An object-oriented refinement calculus for Eiffel. *Form. Asp. Comput.* 16(1), 51–79 (2004)
20. Sekerinski, E.: A type-theoretic basis for an object-oriented refinement calculus. In: *Formal Methods and Object Technology*. Springer, Heidelberg (1996)
21. Stolz, V.: *An integrated multi-view model evolution framework*. Innovations in Systems and Software Engineering (2009)
22. Utting, M.: *An object-oriented refinement calculus with modular reasoning* (1992)
23. van den Berg, J., Jacobs, B.: The loop compiler for java and jml. In: Margaria, T., Yi, W. (eds.) *TACAS 2001*. LNCS, vol. 2031, pp. 299–312. Springer, Heidelberg (2001)
24. von Wright, J.: *Program refinement by theorem prover*. In: *BCS FACS Sixth Refinement Workshop – Theory and Practise of Formal Software Development*. Springer, Heidelberg (1994)

Automating Refinement of *Circus* Programs

Frank Zeyda and Ana Cavalcanti

University of York, Heslington, York, YO10 5DD, U.K.
{zeyda, ana}@cs.york.ac.uk

Abstract. In previous work, we have presented a mechanisation of *Circus* for the theorem prover ProofPower-Z. *Circus* is a refinement language for state-rich reactive systems that combines Z and CSP. In this paper, we present techniques to automate the discharge of proof obligations typically generated by the *Circus* refinement laws. They eliminate most of the proofs that are imposed by the fact that the encoding has to be precise about typing and well-definedness issues, and leave just those that are expected in a pen-and-paper refinement. This allows us to concentrate on the proof of properties that are significant for the problem at hand, while benefiting from the increased assurance and efficiency afforded by the use of a theorem prover as well as high-level tactic languages for refinement. Our case study is a refinement strategy for verification of control systems; we present the result of several experiments.

Keywords: theorem proving, tactics, ArcAngelC, ProofPower.

1 Introduction

Circus [4] is a process algebra that captures state as well as behavioural aspects of a system. Its key concept is that of a *Circus* process, which, like a CSP [15] process, communicates with the environment via channels, but also aggregates local state that can be internally accessed and modified. A *Circus* process specification contains a Z schema that specifies its state, and a list of dependent local actions used by the process, of which a designated action, the main action, defines the process behaviour. Actions may be specified using a mixture of CSP constructs, Z operation schemas [16], and guarded commands of Dijkstra and Morgan [5,8]. Processes can also be combined using CSP constructs.

A notable feature of the *Circus* language is its formal semantics and associated refinement calculus [4]. This permits the derivation or verification of executable programs. The flexibility of the language to handle both sequential behaviour and parallelism in a unified way makes it especially suitable for the description and formal derivation of state-rich concurrent systems [9].

A semantic embedding of *Circus* for ProofPower-Z, a theorem prover based on HOL that also supports Z, was first presented in [12]. We have extended that encoding to handle types and *Circus* programs (rather than the *Circus* semantics [19]). We have also developed a ProofPower implementation of ArcAngelC [18]. This is a refinement-tactic language to formulate strategies to automate the derivation and verification of programs from *Circus* specifications.

A general issue with the mechanisation is the large number of provisos raised by the application of refinement laws. They are mostly conditions that we would not deal with in a pen-and-paper proof. For example, the associativity law $a_1 ; (a_2 ; a_3) \equiv (a_1 ; a_2) ; a_3$ for actions does not require any provisos to be discharged when applied in a pen-and-paper proof. This is not so in its encoding in the semantic embedding of *Circus*, which is given below.

$$\begin{aligned} &\vdash \forall a_1, a_2, a_3 : \text{CIRCUS_ACTION} \mid \\ &\quad \alpha a_1 = \alpha a_2 \wedge \alpha a_2 = \alpha a_3 \bullet a_1 ;_C (a_2 ;_C a_3) \equiv (a_1 ;_C a_2) ;_C a_3 \end{aligned}$$

First, we have to introduce the provisos that a_1 , a_2 and a_3 belong to the set *CIRCUS_ACTION*, the semantic domain for actions. Secondly, two provisos are needed to ensure that all actions have the same alphabet. The α operator gives the alphabet of an action: the variables on which it operates.

These provisos establish well-formedness constraints that are typically taken for granted given the syntactic and type correctness of the actions. A mechanised model, however, forces us to make them explicit to establish the necessary assumptions for provability of the laws. Each *Circus* operator, like sequential composition of actions above, is encoded by a semantic function [12], and these functions are total on *CIRCUS_ACTION*, but only partial with respect to the corresponding maximal set. This gives rise to non-trivial constraints, since *CIRCUS_ACTION*, in our model, is not a type in the sense of being maximal.

In a deep semantic embedding, it is possible to formalise well-formedness as a property of the program syntax. (The difference between a shallow and deep embedding is that the latter also formalises the syntax of the embedded language.) We can then prove that (syntactic) well-formedness implies membership to the aforementioned (semantic) domains. Additionally, a collection of laws may be used to deduce well-formedness of compound expressions from the well-formedness of its components, or conversely, deduce well-formedness of components from the well-formedness of the expression as a whole.

There are, however, important reasons for not pursuing a deep approach in our mechanisation of the UTP and *Circus* to encode alphabetised predicates. The most crucial one is that the language of the UTP is not static: in higher-level theories new operators may be introduced that effectively become part of the syntax. In a deep embedding such extensions would be difficult to handle.

In combination with our implementation of *ArcAngelC* and more complex refinement tactics, the problem of provisos is exacerbated by the fact that tactic application relies on so-called ‘model theorems’, monotonicity of operators, and reflexivity and transitivity of refinement. Those theorems are frequently applied as part of many of the core tactics and suffer from the same problem of introducing conceptually superfluous provisos. Executing the tactics, the provisos quickly accumulate and result in theorems that become unmanageable.

Our contribution is a novel treatment of the notion of well-formedness at the semantic level. It allows us to eliminate most of the inherent provisos in law applications with minimal incisions to the embedding. To take advantage of that, we have made changes to the implementation of *ArcAngelC* as given in [18]; we

also discuss those alterations here. We evaluate our new technique by encoding tactics that perform part of a refinement strategy for control laws [3].

Section 2 introduces the relevant preliminary material: core aspects of our semantic embedding of *Circus*, *ArcAngelC*, and its implementation. Section 3 explains our solution to managing well-definedness, and Section 4 discusses the accompanying extensions to the *ArcAngelC* implementation. Section 5 reports on a case study, and in Section 6 we present our conclusions and future work.

2 Preliminaries

In this section, we first present relevant details of our mechanisation of *Circus*. The mathematical notation we use is standard Z [16], but the ideas in Section 3 and Section 4 also exploit the higher-order support afforded by *ProofPower* being based on HOL. We secondly briefly introduce *ArcAngelC* and its implementation.

2.1 Mechanisation of *Circus*

The mechanisation of *Circus* is based on its denotational semantic model [12], which is formulated in terms of the Unifying Theories of Programming (UTP) [6]. The UTP is a general framework in which the semantics of a variety of modelling and programming languages can be uniformly expressed. It is founded on a relational calculus like Tarski's, but presented in a predicative style.

In the UTP, relations represent computational behaviours. To encode them, we use *alphabetised predicates*, which are predicates equipped with an alphabet of variables. The predicate $x' = x + 1 \vee x' = x - 1$, for example, encodes the computation that either increments or decrements the value of x ; its alphabet includes x and x' . We use undecorated names to denote initial observations of the value of a variable, and dashed names to denote subsequent ones. In the UTP theory for *Circus*, alphabetised predicates describe actions and processes.

In our mechanisation, an alphabetised predicate is a pair.

$$\text{ALPHA_PREDICATE} \hat{=} \{bs : \text{BINDINGS}; u : \text{UNIVERSE} \mid bs \subseteq \text{Bindings}_U u\}$$

Its first component is a set of bindings (records) describing the valuations of the variables of the alphabet that render it true. The second component records the types of the alphabet variables. *BINDINGS* is the type of all binding sets, and *UNIVERSE* consists of all partial functions from *VAR* to *TYPE*, where *VAR* is the semantic domain for variables, and *TYPE* ($\hat{=} \mathbb{P}_1 \text{VALUE}$) contains all non-empty sets of values. The condition $bs \subseteq \text{Bindings}_U u$ effectively establishes that the bindings of a predicate have to be well typed. (The function Bindings_U constructs the complete set of bindings according to a given typing universe.)

Alphabetised predicates are embedded shallowly in the mechanisation: we characterise their semantics, but do not formalise the syntax of the UTP and the *Circus* language. Instead we provide a collection of semantic functions that correspond to the various syntactic constructs.

We define in [19] operators that provide the logical connectives, equality, substitution, including all *Circus* constructs, and importantly refinement. The latter allows us to state that some (concrete) program P behaves according to its

(abstract) specification S . Formally, this is expressed by $S \sqsubseteq P$, and given the mechanisation we can, by aid of a collection of refinement laws, prove whether a given refinement holds. To automate this process, we have implemented a bespoke tactic language `ArcAngelC`; it is explained in the next section.

2.2 ArcAngelC

`ArcAngelC` is a tactic language for the derivation of *Circus* programs from specifications [10]. (Hereafter, we will use the word ‘program’ synonymously for both specifications and programs.) A salient feature of `ArcAngelC` is first that it supports backtracking through angelic choice, namely from failure of tactic applications. This it inherits from its kin `Angel` [7], which is more generally concerned with proving arbitrary goals. Secondly, it has a formal semantics that has been specified in the Z notation, and permits the reasoning about tactics.

Basic literal tactics provided by the `ArcAngelC` are **skip**, which leaves the program unchanged, **fail**, which always fails, and **abort**, whose application is not guaranteed to terminate. To apply refinement laws, the tactic **law name**(*args*) takes the name of the law and a list of arguments. Compound tactics may be declared using the **Tactic name**(*args*) $\hat{=}$ *body* **end** construct.

We further have the binary tacticals $t_1 ; t_2$ for sequence and $t_1 \mid t_2$ for alternation. Sequence executes the tactics one after another, and alternation first attempts to apply t_1 , and if that fails, applies t_2 . Other tactics are $!t$ which acts like a cut on the backtracking search of finding a successful path of tactic execution, and **applies to** p **do** t which guards the application of a tactic t by successful matching of the program against a pattern p . Finally, recursive tactics are supported via the fixed-point operator $\mu X \bullet t(X)$.

To apply tactics to the operands of a *Circus* action or process construct, a set of structural combinators is provided. They are boxed versions of the respective *Circus* operators. For example, the combinator $t_1 \boxed{\square} t_2$ applies to actions of the form $a_1 \square a_2$. Its behaviour is to apply t_1 to a_1 and t_2 to a_2 . The application is justified by the monotonicity of *Circus* operators with respect to refinement.

We have implemented `ArcAngelC` in `ProofPower` [18]. The fundamental design of the implementation supports tactics as theorem-generating functions that apply to program expressions A and return lists of refinement theorems $\Gamma \vdash A \sqsubseteq B$. The construction of refinement theorems is necessarily sound because of the LCF approach that prevents invalid theorems from being derived. More specifically, `ProofPower-Z` uses the type system of the prover’s implementation language (ML) to differentiate between (unproved) conjecture and (proved) theorems.

3 Managing Well-Definedness

Most of the laws of our *Circus* semantic encoding specify well-definedness provisos for their applicability. The provisos ensure that relational expressions, such as $p_1 \sqsubseteq p_2$, as well as operator applications, like $p_1 \square_C p_2$, are well-defined, that is, the underlying semantic functions are applied inside their domain.

As already said, these well-definedness constraints give rise to proof obligations that accumulate through the application of `ArcAngelC` tactics. Sources for the provisos are model theorems automatically applied in the mechanics of the `ArcAngelC` implementation, monotonicity theorems applied when invoking structural combinators, and user-defined laws. In practice, even after applying just the first three of the seven phases of the refinement tactic `NB` for control laws [13], the resulting theorem already includes more than 130 assumptions. Thereafter it becomes unmanageable, slowing down or even bringing to a stall further application of tactics. To tackle this problem we have pursued a combination of two approaches. They are explained separately in the following sections.

3.1 Reducing Constraints in the Semantic Encoding

To tame the complexity of generated assumptions, we have used a novel treatment of typing. This reduced the number of provisos but has retained soundness.

By way of illustration, the semantic function for \wedge is defined as follows.

$$\left| \frac{(- \wedge_P -) : WF_ALPHA_PREDICATE_PAIR \rightarrow ALPHA_PREDICATE}{\forall p_1, p_2 : ALPHA_PREDICATE \mid (p_1, p_2) \in WF_ALPHA_PREDICATE_PAIR \bullet p_1 \wedge_P p_2 = (t_B, t_U)} \right|$$

Here, membership of (p_1, p_2) to `WF_ALPHA_PREDICATE_PAIR` encapsulates an additional constraint for the compatibility of the universes of p_1 and p_2 . The terms t_B and t_U respectively abbreviate the binding set and universe of the result; their particular shape is not relevant here and for brevity is omitted.

Opposed to this, in `HOL`, the types of variables are part of their identity, meaning that in a term like $n \geq 1 \wedge n = \mathbf{false}$ (that may result from conjoining $\Gamma_1 \vdash n \geq 1$ and $\Gamma_2 \vdash n = \mathbf{false}$), we are effectively talking about two different variables n distinguished by their types. We adopt a similar approach by moving type information from the universe directly into the entities representing names, which are now bindings of the following schema type.

$$VAR \hat{=} [name : STRING; dashes : \mathbb{N}; subscript : SUBSCRIPT; type : TYPE]$$

The type `STRING` represents character sequences, and `SUBSCRIPT` is a free type for representing a possible subscript. Importantly, the `type` component records the type of the variable, and `TYPE` is equated with the non-empty subsets of `VALUE`, that is, $TYPE \hat{=} \mathbb{P}_1 \text{ VALUE}$. In comparison, in the previous model, `VAR` only recorded a unique identifier (name), dashes, and a subscript.

This implies that the previous notion of ‘universe’ is subsumed by the conventional notion of an alphabet, which now implicitly carry type information. The encoding we obtain is in fact very similar to that originally developed by Oliveira [12], but with the added benefit of concisely capturing type information. The constraint $bs \subseteq Bindings_U \ u$ is notably redundant now in the definition of `ALPHA_PREDICATE`. We can also drop additional constraints in definitions that require compatibility of types. Thus, the definition of \wedge_P becomes

$$\left| \frac{(_ \wedge_P _) : ALPHA_PREDICATE \times ALPHA_PREDICATE \rightarrow \dots}{\forall p_1, p_2 : ALPHA_PREDICATE \bullet p_1 \wedge_P p_2 = (t_B, t_A)} \right.$$

Most other definitions of our encoding can be simplified in a similar manner. The notion of compatibility becomes obsolete since the common variables of alphabetised predicates necessarily have the same type.

This enhancement also displayed benefits in terms of modularising proofs. First, reasoning about alphabets (which are sets) is logically simpler than reasoning about universes (which are functions). Secondly, it is now possible to introduce a notion of well-typed expressions independently of the universe context; this makes provisos related to evaluation of expressions simpler.

We, however, still have many provisos like $p \in ALPHA_PREDICATE$ or $p \in CIRCUS_ACTIONS$. The following section explains how we deal with them.

3.2 A Semantic Formalisation of Well-Formedness

Our approach to capture well-formedness at the semantic level gives us the same benefits as a syntactic characterisation of well-formedness in a deep embedding, and importantly does not compromise soundness. To formalise well-formedness, we introduce a generic HOL function wd of type $'a \rightarrow BOOL$ where $'a$ is a type variable. (Hereafter we use the term ‘well-defined’ in favour of ‘well-formed’.) Initially we do not specify any properties of wd , but for each semantic operator op we add an axiomatic constraint of the following form.

$$\vdash wd(op(x_1, \dots, x_n)) \Leftrightarrow wd\ x_1 \wedge \dots \wedge wd\ x_n \wedge (x_1, \dots, x_n) \in \text{dom } op$$

The proposition $wd(op(x_1, \dots, x_n))$ entails that op is applied in its domain, and that all arguments are well-defined. Domain membership enables us to extract properties of the arguments; for instance $wd(p_1 \wedge_P p_2)$ implies that $p_1 \in ALPHA_PREDICATE$. Moreover, for complex program terms, the property of well-defined arguments allows us to extract well-definedness of any sub-term. For example, $wd(p_1 \wedge_P (p_2 \wedge_P p_3))$ implies $wd\ p_1$, $wd\ p_2$ and $wd\ p_3$.

A potential risk with this approach is due to the definition of wd not being obviously conservative. Generally in logic, conservative extensions maintain consistency of the extended theory, and in many case this can be established by the mere shape of the defining axiom. What we are doing, however, in defining wd is in fact treating some Z function application $op(x_1, \dots, x_n)$ as if it revealed information whether the function was applied in its domain. It is not immediately evident if and under what conditions such a treatment is sound.

To prove consistency, we provide a model which fulfils the defining axioms. The essence of our model is the identification of values outside the domain and range of each operator with undefinedness, and axioms that constrain applications of operators outside their domains to be closed under this set. This is possible because the underlying HOL logic is based on total functions, and therefore any term denotes a value. Specifically, $f(x)$ denotes a value even when $x \notin \text{dom } f$, and constraining this value by an axiom such as $\vdash f(c) = v$ for particular c and

v neither impinges on f 's domain nor produces unsoundness if $c \notin \text{dom } f$. This is a property of the embedding of \mathbb{Z} partial functions into HOL .

To illustrate the conceptual idea of the model, we consider, for example, the set $ALPHA_PREDICATE$. To capture undefinedness for functions encoding alphabetised predicate operators, we introduce the following set.

$$\perp_{ALPHA_PREDICATE} \hat{=} \mathbb{U} \setminus ALPHA_PREDICATE$$

It is simply the complement of $ALPHA_PREDICATE$ with respect to its corresponding maximal type. In ProofPower-Z , \mathbb{U} acts as the carrier set of a generic type which is inferred by the type checker, and which is always maximal. Here, it would be the set $\mathbb{P}((\mathbb{P} \text{VAR}) \times \mathbb{P}(\text{VAR} \leftrightarrow \text{VALUE}))$.

For \wedge_P , for example, we have the supplementary axiom below.

$$\vdash \forall p_1, p_2 : \mathbb{U} \mid (p_1, p_2) \notin \text{dom}(_ \wedge_P _) \bullet p_1 \wedge_P p_2 \in \perp_{ALPHA_PREDICATE}$$

It does not affect (relative) consistency because none of the original definitions impose any constraints on function applications outside their domains.

Finally, we can give a conservative definition for wd if applied to elements of type $ALPHA_PREDICATE$: $wd \ p \Leftrightarrow p \notin \perp_{ALPHA_PREDICATE}$. The definition provably satisfies the axiom for $wd(p_1 \wedge_P p_2)$ as it has been specified before, and thereby establishes the correctness of the model, which itself is sound.

This model in a way simulates a treatment of undefinedness. It is correct if the types are ‘large enough’ so that we can always find a witness that serves to distinguish defined from undefined function applications.

There are cases where a collection of semantic types T_1, T_2, \dots have the same maximal type T_{max} . For example, the semantic domain $CIRCUS_ACTION$ for actions is a subset of $ALPHA_PREDICATE$. In those cases, a single set \perp_T is defined as $T_{max} \setminus \bigcup T_i$ to ensure that \perp_T is disjoint from all sets T_i . In such a situation, the union $\bigcup T_i$ needs to be a proper subset of T_{max} , so that there is some $x \in T_{max}$ for which $\forall i \bullet x \notin T_i$. This implies that $\perp_T \neq \emptyset$, which is crucial to prove that the model satisfies the axioms for wd .

Accordingly, there are functions in our encoding to which we cannot apply wd . These are first the operators involving the ProofPower-Z \mathbb{B} type, which is maximal. Since refinement is defined by a function with range \mathbb{B} , we cannot specify $wd(p_1 \sqsubseteq p_2) \Leftrightarrow wd \ p_1 \wedge wd \ p_2 \wedge (p_1, p_2) \in \text{dom}(_ \sqsubseteq _)$. The impact of this restriction is on provisos that involve refinements themselves. The extension of our technique to handle such cases is left as future work.

Additionally, the domains of the functions that encode the various operators on values is $VALUE$, which is also maximal. This prevents us from specifying a well-definedness axiom, for example, for $Eval(b, e)$, which evaluates an expression under a binding and yields an element of $VALUE$. To handle expressions, we axiomatise wd slightly differently; we define wd inductively over the free type $EXPRESSION$ that encodes the syntax of expressions. (Unlike alphabetised predicates, they are embedded deeply.) We introduce a set $WT_EXPRESSION$ containing the expressions that are well-defined, that is $\{e : EXPRESSION \mid wd \ e\}$. Since, the functions that encode *Circus* operators

involving expressions are parameterised in terms of *WT_EXPRESSION*, their domains are not maximal, and so we can give *wd* axioms for them.

We do not actually define our model for *wd* in *ProofPower-Z*, as it would unnecessarily complicate the various definitions of operators. The main point is that we can introduce *wd* capturing the well-formedness of terms purely in semantic terms, and given the above caveats this definition is sound.

It is now possible to specify laws that either exploit or prove *wd* theorems. In particular, the associativity law for conjunction is now expressible as below.

$$\forall p_1, p_2, p_3 : \mathbb{U} \mid wd((p_1 \wedge_P p_2) \wedge_P p_3) \bullet \\ (p_1 \wedge_P p_2) \wedge_P p_3 \equiv p_1 \wedge_P (p_2 \wedge_P p_3) \wedge wd(p_1 \wedge_P (p_2 \wedge_P p_3))$$

This directly mirrors the intuition that if the left-hand side $p_1 \wedge_P (p_2 \wedge_P p_3)$ is well-defined, the equivalence holds and also the right-hand side $(p_1 \wedge_P p_2) \wedge_P p_3$ is well-defined. Its only non-trivial proviso is the assumption of well-definedness of the initial program. We mechanically proved the above law without much effort by utilising its original version and rewriting applications of *wd*.

This illustrates how *wd* can provide a framework in which we can handle the emerging proof obligations for typing with theorems that establish that well-formedness is preserved. We work in a setting similar to that of pen-and-paper refinement proofs, where we normally assume that the initial program is well-formed, as are the programs of each law. If additionally the arguments of parameterised laws are well-formed, we conclude that all programs in the derivation chain must be well-formed. In summary, we work under assumptions that mean that we do not need to worry about issues of well-formedness.

In the following section we explain how the implementation of *ArcAngelC* has been amended to make use of theorems that possess this shape.

4 Extensions to the *ArcAngelC* Implementation

We present here how we have extended the *ArcAngel* implementation to take advantage of the *wd* function.

4.1 Extended Refinement Theorems

Refinement theorems in our implementation of *ArcAngelC* had to be of the form $\Gamma \vdash A \sqsubseteq B$, where Γ is a list of proof obligations, and A and B are program expressions. To integrate well-definedness constraints, we have generalised the permissible shape of such theorems to $\Gamma, wd \ A \vdash A \sqsubseteq B \wedge wd \ B$. We call them *extended refinement theorems*. Their conclusions are conjunctions in which the first conjunct provides the actual refinement, and the second conjunct establishes well-definedness of the result B of the program transformation. We also have an assumption that asserts well-definedness of the initial program A .

The implementation has been adapted to handle these kinds of theorems. Importantly, the *ArcAngelC* mechanics has been adjusted as to only retain the *wd* theorem of the initial program, and, as much as possible, discard intermediate *wd*

provisos via incremental proofs. We also importantly preserve the general shape of an extended refinement theorem during tactic applications. To illustrate this, we consider the application of a refinement law. In the previous encoding these laws were typically of the form

$$\vdash \forall v_1 : T_1; \dots; v_n : T_n \mid P[v_1, \dots, v_n] \bullet A[v_1, \dots, v_n] \sqsubseteq B[v_1, \dots, v_n]$$

with type provisos $v_1 : T_1$, $v_2 : T_2$, and so on. The notation $A[v_1, \dots, v_n]$ is used to emphasise that the variables v_i are free in A . We now rephrase such laws as

$$\begin{aligned} \vdash \forall v_1 : \mathbb{U}; \dots; v_n : \mathbb{U} \mid wd\ A[v_1, \dots, v_n] \wedge \\ P[v_1, \dots, v_n] \bullet A[v_1, \dots, v_n] \sqsubseteq B[v_1, \dots, v_n] \wedge wd\ B[v_1, \dots, v_n] . \end{aligned}$$

Here, \mathbb{U} is the carrier set of a type that is dynamically inferred by the type checker. It is maximal, and so only incurs trivial proof obligations. This new version of the law can be proved from the former by rewriting the *wd* function on both sides and extracting the type provisos. For example, if A is $v_1 \wedge_P v_2$ we have that $wd(v_1 \wedge_P v_2)$ implies that both v_1 and v_2 belong to *ALPHA_PREDICATE*.

We observe that all type provisos have been absorbed into just one assertion $wd\ A[v_1, \dots, v_n]$ that establishes well-definedness of the left-hand program. If we apply the new law to an extended refinement theorem $\Gamma, wd\ X \vdash X \sqsubseteq Y \wedge wd\ Y$, the right-hand program Y is matched against the left-hand program A of the law to instantiate the quantified variables. We then obtain an instantiation $P', wd\ Y \vdash Y \sqsubseteq Y' \wedge wd\ Y'$ after moving antecedents to the hypotheses.

The original extended refinement theorem and the instantiated law give rise to the following four theorems (after eliminating the conjunction in the conclusions): (1) $\Gamma, wd\ X \vdash X \sqsubseteq Y$; (2) $\Gamma, wd\ X \vdash wd\ Y$; (3) $P', wd\ Y \vdash Y \sqsubseteq Y'$; (4) $P', wd\ Y \vdash wd\ Y'$. We use (1), (3) and (4) to obtain

$$(5) \Gamma, P', wd\ X, wd\ Y \vdash X \sqsubseteq Y' \wedge wd\ Y'$$

by transitivity of refinement using (1) and (3), and conjunction using (4). The transitivity model theorem is now recast as

$$\vdash wd\ p_1 \wedge wd\ p_2 \wedge wd\ p_3 \bullet p_1 \sqsubseteq p_2 \wedge p_2 \sqsubseteq p_3 \Rightarrow p_1 \sqsubseteq p_3 .$$

It is formulated in terms of *wd* theorems rather than type provisos.

The intermediate theorem (5) contains the surplus assumption $wd\ Y$, and our goal is to eliminate it. This can be achieved by the cut rule together with (2). The cut rule states that from $\Gamma_1 \vdash P$ and $\Gamma_2, P \vdash Q$ we can derive $\Gamma_1, \Gamma_2 \vdash Q$. All this requires no real proof effort and can be done with the application of rules. (Rules are theorem-generating functions in *ProofPower*, and their evaluation usually requires less effort.) We then obtain the extended refinement theorem

$$(6) \Gamma, P', wd\ X \vdash X \sqsubseteq Y' \wedge wd\ Y'$$

with the desired shape. Apart from the *wd* assertion for the initial program and provisos Γ , it only introduces the relevant proof obligations P' of the law.

All this takes place as part of the mechanics of the implementation of law applications. It relies, however, on the model and law theorems having the correct

shape. For instance, if the law does not have the *wd* term of the transformed program in its consequent, the elimination of the proviso in (5) fails. The implementation is, nonetheless, sufficiently robust to handle such cases; this merely shows in the accumulation of provisos that could not be removed.

The examples used so far do not illustrate how provisos in the recast laws reduce those of the former laws beyond typing conditions. For example, assignment is encoded by the semantic function $Assign_C(a, ns, es)$ where a is the alphabet, and ns and es are sequences of variables and expressions. Here, $wd\ Assign_C(a, ns, es)$ does not merely constrain a to be of type $ALPHABET$, ns to be of type $seq\ VAR$, and es to be of type $WT_EXPRESSION$, but states the stronger $(a, ns, es) \in \text{dom } Assign_C$, which, by definition of $Assign_C$, moreover implies that the sequences ns and es have the same length. Conditions like these would formerly have to be explicitly specified in the antecedent of the law. The simplification becomes even more apparent when the left and right-hand programs of the law are more complex. Despite, experience so far shows that in certain cases we still need to specify non-trivial type provisos, namely when in a law $A \sqsubseteq B$ we cannot prove $wd\ B$ from $wd\ A$, but this is not very often the case.

The next section discusses an approach to the treatment of provisos similar to the one above, but in the context of monotonicity theorems. It ensures that the application of structural combinators does not introduce further assumptions.

4.2 Structural Combinators

The *ArcAngelC* implementation requires, for each combinator, two monotonicity theorems: one for equivalence and one for refinement. These theorems are used when applying the respective combinator tactic. They also give rise to provisos that accumulate. In general, the monotonicity theorems are of the form

$$\begin{aligned} & \vdash \forall a_1 : T_1; \dots; a_k : T_k; p_1, \dots, p_n : T; p'_1, \dots, p'_n : T \mid \\ & P[a_1, \dots, a_k, p_1, \dots, p_n] \wedge P'[a_1, \dots, a_k, p'_1, \dots, p'_n] \bullet \\ & p_1 \sqsubseteq p'_1 \wedge p_2 \sqsubseteq p'_2 \wedge \dots \wedge p_n \sqsubseteq p'_n \Rightarrow \\ & \mathbf{op}(a_1, \dots, a_k, p_1, \dots, p_n) \sqsubseteq \mathbf{op}(a_1, \dots, a_k, p'_1, \dots, p'_n) \end{aligned}$$

where $\mathbf{op}(a_1, \dots, a_k, p_1, \dots, p_n)$ is an $(n+k)$ -ary operator with fixed arguments a_1, \dots, a_k and monotonic arguments p_1, \dots, p_n . For example, a conditional is a programming operator whose condition is a fixed argument, and whose programs are monotonic arguments. For action operators, T is $CIRCUS_ACTION$. The provisos P and P' establish that the application of \mathbf{op} is well-defined on both sides of the concluding refinement. They imply that $(a_1, \dots, a_k, p_1, \dots, p_n)$ and $(a_1, \dots, a_k, p'_1, \dots, p'_n)$ both belong to $\text{dom } \mathbf{op}$.

Using the *wd* function, we generally express these theorems now as

$$\begin{aligned} & \vdash \forall a_1 : \mathbb{U}; \dots; a_k : \mathbb{U}; p_1, \dots, p_n : \mathbb{U}; p'_1, \dots, p'_n : \mathbb{U} \mid \\ & wd(\mathbf{op}(a_1, \dots, a_k, p_1, \dots, p_n)) \wedge p'_1 \in T \wedge \dots \wedge p'_n \in T \bullet \\ & p_1 \sqsubseteq p'_1 \wedge p_2 \sqsubseteq p'_2 \wedge \dots \wedge p_n \sqsubseteq p'_n \Rightarrow \\ & \mathbf{op}(a_1, \dots, a_k, p_1, \dots, p_n) \sqsubseteq \mathbf{op}(a_1, \dots, a_k, p'_1, \dots, p'_n) \wedge \\ & wd(\mathbf{op}(a_1, \dots, a_k, p'_1, \dots, p'_n)) \ . \end{aligned}$$

This shape is similar to that of rephrased laws, but additionally includes the provisos $p'_i \in T$ for all $i \in \{1, \dots, n\}$. We explain below how they are discarded during the application of the tactic. We use $\boxed{\square}$, the structural combinator for internal choice, as an example. The approach applies to all unary, binary and n-ary combinators of the ArcAngelC program model. An exception is the combinator $\boxed{\mu}$ for recursion, which we address separately.

The combinator $\boxed{\square}$ for *Circus* internal choice has the following monotonicity theorem in our original implementation. (There are no fixed arguments.)

$$\begin{aligned} \vdash \forall p_1, p_2, p'_1, p'_2 : \text{CIRCUS_ACTION} \mid \alpha p_1 = \alpha p_2 \bullet \\ p_1 \sqsubseteq p'_1 \wedge p_2 \sqsubseteq p'_2 \Rightarrow p_1 \sqcap_C p_2 \sqsubseteq p'_1 \sqcap_C p'_2 \end{aligned}$$

Besides the type provisos, we require with $\alpha p_1 = \alpha p_2$ that p_1 and p_2 have the same alphabet. This is crucial for the well-definedness of $p_1 \sqcap_C p_2$. The refinements in the antecedent imply that $\alpha p_1 = \alpha p'_1$ and $\alpha p_2 = \alpha p'_2$, hence we can conclude that $\alpha p'_1 = \alpha p'_2$ holds too. This ensures well-definedness of $p'_1 \sqcap_C p'_2$. Accordingly, the new monotonicity theorem is as follows.

$$\begin{aligned} \vdash \forall p_1, p_2 : \mathbb{U}; p'_1, p'_2 : \mathbb{U} \mid wd(p_1 \sqcap_C p_2) \wedge \\ p'_1 \in \text{CIRCUS_ACTION} \wedge \\ p'_2 \in \text{CIRCUS_ACTION} \bullet \\ p_1 \sqsubseteq p'_1 \wedge p_2 \sqsubseteq p'_2 \Rightarrow p_1 \sqcap_C p_2 \sqsubseteq p'_1 \sqcap_C p'_2 \wedge wd(p'_1 \sqcap_C p'_2) \end{aligned}$$

We observe that we cannot rid ourselves of all type provisos: membership of p'_1 and p'_2 to *CIRCUS_ACTION* stays, because refinement of p_1 and p_2 does not necessarily preserve membership to *CIRCUS_ACTION*. We can, however, use a proof tactic to remove these provisos in the resulting theorem by proving them from the residual assumptions and again using the cut rule as follows.

When a structural combinator is applied, the first step is to dissect the operator application and apply the combinator tactics to the program operands. Here, for example, applying $t_1 \boxed{\square} t_2$ to $A \sqcap_C B$ applies t_1 to A and t_2 to B . This results in the refinement theorems $\Gamma_1, wd A \vdash A \sqsubseteq A' \wedge wd A'$ and $\Gamma_2, wd B \vdash B \sqsubseteq B' \wedge wd B'$. We then conjoin these theorems to obtain $\Gamma_1, \Gamma_2, wd A, wd B \vdash A \sqsubseteq A' \wedge B \sqsubseteq B'$ whose conclusion has now the right shape to apply the monotonicity theorem in a forward-chaining manner, that is, using modus ponens to obtain the conclusion of the theorem. This yields

$$\begin{aligned} \Gamma_1, \Gamma_2, A' \in \text{CIRCUS_ACTION}, B' \in \text{CIRCUS_ACTION}, \\ wd A, wd B, wd(A \sqcap_C B) \vdash A \sqcap_C B \sqsubseteq A' \sqcap_C B' \wedge wd(A' \sqcap_C B') \end{aligned}$$

From the application of wd to the original program, in our example $A \sqcap_C B$, it follows, by the definition of wd , that the operands are well-defined; in our example, $wd A$ and $wd B$ hold, so both can be easily eliminated as before. For the elimination of the assumptions related to the restrictions on the components of the new program, that is, $A' \in \text{CIRCUS_ACTION}$ and $B' \in \text{CIRCUS_ACTION}$ in the example above, we exploit the wd theorems of the individual program refinements. They are in the case above $\Gamma_1, wd A \vdash wd A'$ and $\Gamma_2, wd B \vdash wd B'$.

A proof tactic uses these theorems by rewriting $wd\ A'$. In the example above, for a binary structural combinator, there are two terms $wd\ A'$ and $wd\ B'$. In the case of a unary combinator, there is only one, and so on. If A' is an application $op_C(A'')$ of a *Circus* operator op_C , then $wd\ A'$ yields that $A'' \in \text{dom } op_C$. A simple law can then be used to infer that $op_C(A'') \in \text{ran } op$. Since, the range of *Circus* operators is usually *CIRCUS_ACTION*, we obtain a proof for $A' \in \text{CIRCUS_ACTION}$. Considering such results for all terms $wd\ A'$, $wd\ B'$, and so on, is usually enough to establish well-definedness of the new program.

We observe that this reasoning does not depend on the structural combinator *per se*. The assumptions are that the provisos are exclusively of the form $p'_i \in \text{CIRCUS_ACTION}$, that there exists a wd axiom for every *Circus* operator, and that all *Circus* operators have domains *CIRCUS_ACTION*.

In our example, this permits us to eliminate the two other provisos too.

$$\Gamma_1, \Gamma_2, wd(A \sqcap_C B) \vdash A \sqcap_C B \sqsubseteq A' \sqcap_C B' \wedge wd(A' \sqcap_C B')$$

The result contains only the genuine proof obligations, so no overhead is incurred by using the monotonicity rule in the mechanisation of the combinator.

The crucial point here is that in the case of structural combinators some real proof effort is needed to eliminate the surplus provisos. Our implementation of *ArcAngelC* is in fact an instantiation of a framework that we have developed for angelic languages for refinement tactics. It supports *ArcAngelC* and its predecessor *ArcAngel* [11] for Morgan's refinement calculus, and can be extended for other languages of the same nature. To manage structural combinators, we have extended the framework to support the dynamic configuration of tactics used to discharge provisos emerging from their application. For all *ArcAngelC* combinators, the tactics we define are sufficient. The provided flexibility, however, permits the inclusion of custom tactics that may be needed for other languages.

There are cases in which elimination of the assumptions fails, namely, if the arguments of the operator are not of the form $op_C(\dots)$ where op_C is some *Circus* operator. Experience, however, shows that this does not happen very often.

More specific extensions have been necessary to handle the structural combinator $\overline{\mu}$ for recursion, which, unlike the other combinators, is supplied with a function on alphabetised predicates, usually given as a λ -expression. The proof steps for removing provisos are more involved, and in fact it was not possible to remove all of them due to the fact that pointwise refinement of a function does not preserve monotonicity of the function. Precisely, from monotonicity of f ($\forall x, y \in \text{dom } f \bullet x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$), and the pointwise property $\forall x \in \text{dom } f \bullet f(x) \sqsubseteq f'(x)$ we cannot conclude that f' is monotonic too. In pen-and-paper proofs, we justify monotonicity assumptions by the particular shape of actions only using monotonic constructs. In a shallow embedding, capturing this is more difficult because we cannot give a general theorem stating that *Circus* actions are monotonic. Solving this problem is part of ongoing work.

The next section comments on experiences in the context of a case study.

5 Practical Experiences

The primary motivation for the treatment of well-definedness in the last two sections is the automation of complex refinement strategies using our tools and the semantic embedding of *Circus*. As a case study, we have automated part of the refinement strategy for control laws described in [2,3].

The work in [2,3] describes first how a formal account of (a subset of) the Simulink notation can be given by translating the diagrams into *Circus* specifications. (Simulink is a *de facto* industry standard for the design of control systems.) It then presents a strategy for refinement that gradually transforms the diagram specification into a *Circus* model of a given Ada program, and thereby verifies the correctness of the Ada implementation. For the encoding of the ArcAngelC tactics, we refer to the work in [13]. The tactics we mechanise are mostly literal translations of those in [13], including the various *Circus* laws they rely upon.

The refinement strategy is structured into four phases, and so far we only automated the first phase NB. This is itself subdivided into 7 steps, NBStep1 to NBStep7, which are assembled into one compound tactic NBMain. The tactic deals with the normalisation of *Circus* processes representing blocks (or subsystems) that are realised sequentially in code. Their model is given by a single centralised process with two parallel actions. The tactic attempts to remove the parallelism between these actions. A more detailed account of the refinement strategy is omitted as it would take us too far from the agenda of the paper.

Our example specification is part of the model for the PID controller in [3]. In particular, we have applied the tactics to the model of a differentiator. It is a simple example, which, however, as we discuss below, already reveals the importance and effectiveness of our treatment of well-formedness.

Having encoded the tactics and laws, we have verified by inspection, that all assumptions that remain after the application of the tactics are genuine proof obligations. As explained in Section 4.2, particular kinds of provisos, namely those resulting from $\boxed{\mu}$, at present cannot be discharged automatically. We verified, however, that these are the *only* residual provisos, apart from the well-definedness assumption of the initial program of the refinement.

This initial analysis has uncovered some glitches in the recast theories and implementation that have been subsequently fixed. One of them was a missing *wd* axiom for one of the *Circus* operators which resulted in a simplification tactic failing. As already mentioned, the implementation has been designed to robustly deal with such cases, hence no error is raised when individual simplifications of provisos do not succeed. (In some cases this leads to a partial proof encoded as a ‘reduced’ proviso, and in other cases the proviso remains unchanged.) We have increased the efficiency of simplification tactics by incorporating guarding conditions that ensure they are only applied to assumptions of the correct shape, and furthermore only involve a predictably small number of proof steps.

We have also compared the efficiency of our technique to that of the standard explicit treatment of provisos. Since our extensions to the ArcAngelC implementation are fully backward compatible, it is possible to revert to the old shape of laws and model theorems and execute the same tactics under the same

Phase	POs Before	POs Current	Run-time Before	Run-time Current
NBStep1	1	1	0.0 sec	0.1 sec
NBStep2	44	4	2.3 sec	1.1 sec
NBStep3	101	8	9.1 sec	2.4 sec
NBStep4	163	15	17.9 sec	3.9 sec
NBStep5_6	198	18	19.9 sec	4.7 sec
NBStep7	243	24	21.1 sec	5.9 sec

Fig. 1. Comparison of the number of generated provisos for NB

conditions. Table 1 summarises the results obtained for the tactics NBStep1 to NBStep7 of the refinement strategy. (NBStep5 and NBStep6 have been merged into one tactic). Each row entails the invocation of the preceding tactics too; for instance, the second row refers to the execution of both NBStep1 and NBStep2. We report on the number of remaining provisos as well as the execution time of the tactics measured by the timing facilities of Poly/ML.

The results show that assumptions are reduced by approximately 90%. This remains fairly constant across tactics, and invariant with the growth in size. We may expect a similar reduction in runtime, but it is offset by the effort required to discharge the provisos. Despite, we see that overall this effort is recovered, and the gain increases with the complexity of the generated refinement theorems. For example, after NBStep2 we only have a small speed-up, and for NBStep1 even a slight loss. The speed-up, however, becomes larger in the next two phases.

To check if this trend continues with more complex examples, we have slightly amended the Diff process. This has resulted in larger ProofPower-Z terms and the need for more elementary steps to transform the program. In this case, the original implementation takes 61sec to execute phases NBStep1 to NBStep5_6 and generates a theorem with 259 assumptions. In comparison, the new technique requires 7.2sec producing 24 assumptions, giving a further speed-up increase to 8.5. After once more elaborating the complexity of the example specification, the former approach requires 154.4sec and produces 301 assumptions whereas the new ArcAngelC implementation only takes 10.9 sec. Although we still have a similar 90% reduction in assumptions, the speed-up has now increased to 14.1.

The above suggests that with growing complexity of the ArcAngelC tactics and *Circus* specifications, the speed-up may further increase, even so non-linearly. Reasons for this may be that certain operations on theorems are slowed down in a non-proportional way when theorems become larger. It should be pointed out that the system we compare with already includes the simplification to the mechanisation reducing type provisos that we discussed in Section 3.1.

Our results increase confidence that, with the new technique and tool support, it is possible now to apply the entire refinement strategy and obtain a result within a reasonable amount of time.

6 Conclusions

We have presented a treatment of well-formedness in a shallow embedding of *Circus* that considerably simplifies the provisos generated in the application of refinement (and other) laws. Although it was described in the context of our *Circus* mechanisation, the techniques and results obtained generalise to other language embeddings in other tools that raise similar issues.

We have introduced the *wd* function to capture well-formedness of terms *as if* they were syntactic rather than semantic entities. We have then identified constraints to establish that this axiomatisation, though non-conservative, is sound. We have thus avoided the added complexity of a deep embedding while reaping some of its benefits in relation to properties that are inherently syntactic.

The underlying model for *wd* is actually a strict treatment of undefined values. We, however, do not make its model explicit and content ourselves with its mere existence. Since the logic of HOL does not accommodate undefinedness, limitations arise in that we cannot, for instance, utilise *wd* on boolean functions because the type \mathbb{B} is not ‘big enough’ to provide enough values to represent the undefined case. This is a trade off that we have to accept.

The *wd* function crucially paves the way for the efficient application of more complex refinement tactics as it enabled us to recast the implementation of *ArcAngelC* in such a way that, apart from genuine proof obligations, only one additional proviso is generated — to establish the well-definedness of the initial program. This keeps the complexity of emerging refinement theorems at bay, and thereby creates opportunities for the use of our mechanisation of *Circus* and tools for automatic refinement in the context of real industrial systems.

In [17], von Wright presents a tool for stepwise refinement in a simple sequential command language. Its semantics is characterised in terms of wp predicate transformers, which are shallowly embedded into HOL. Well-definedness conditions loosely correspond to establishing the monotonicity of predicate transformers, however the semantic domain is not *a priori* restricted to those.

The implication of von Wright’s and similar approaches are that we required more relaxed definitions of operators, and also accept limitations on what laws can be *generally* established for the semantic entities. In the UTP, we often rely on proving properties from healthiness conditions rather than by induction over some syntax, which makes our approach more suitable here.

Other related work is the refinement editor *Refine* and *Gabriel* [14] which supports the specification and interactive application of *ArcAngel* tactics to derive programs in Morgan’s calculus. These tools notably offer facilities to interactively apply tactics and laws. Otherwise, *Refine* and *Gabriel* were developed in view of a specific language, and are essentially rewrite systems.

More recent work has been done on developing *Saoithín* [1], a proof assistant specifically designed for the UTP. Its advantages are that it essentially operates at the level of syntax, and naturally some of the semantic issues we reported on do not arise. It also provides proof strategies that are optimised for proofs in

the UTP. On the other hand, it does not specify a model for its deductive system and calculus, and this imposes limitations on proving consistency of theory extensions. An interesting line of work could be to try and combine Saoithín with our tools to see if we can reap individual benefits of both of them.

Future work will first consist of mechanising the entire refinement strategy for control laws in [13]. Secondly, further work is required to provide proof automation in relation to the well-definedness of the initial program. To obtain unqualified theorems, those assumptions need to be discharged. It is still an open issue how provisos in such theorems are best proved, and whether some of the *wd* theorems that we discard as we go along should be cached for later use.

Acknowledgements. This work has been funded by EPSRC as part of the Programming from Control Laws research grant EP/E025366/1.

References

1. Butterfield, A.: Saoithín Proof Assistant, <http://www.scss.tcd.ie/Andrew.Butterfield/Saoithin/>
2. Cavalcanti, A., Clayton, P., O'Halloran, C.: Control Law Diagrams in Circus. In: FM 2005: Formal Methods. LNCS, vol. 3582, pp. 253–268. Springer, Heidelberg (2005)
3. Cavalcanti, A., Clayton, P., O'Halloran, C.: From Control Law Diagrams to Ada via Circus. Technical report, University of York, York, U.K. (April 2008)
4. Cavalcanti, A., Sampaio, A., Woodcock, J.: A Refinement Strategy for Circus. *Formal Aspects of Computing* 15(2-3), 146–181 (2003)
5. Dijkstra, E.: *A Discipline of Programming*. Prentice Hall Series in Automatic Computation. Prentice Hall, Englewood Cliffs (1976)
6. Hoare, C.A.R., Jifeng, H.: *Unifying Theories of Programming*. Prentice Hall Series in Computer Science. Prentice Hall, Englewood Cliffs (1998)
7. Martin, A., Gardiner, P., Woodcock, J.: *A Tactic Calculus - Abridged Version*. *Formal Aspects of Computing* 8(4), 479–489 (1996)
8. Morgan, C.: *Programming from Specifications*. Prentice Hall International Series in Computer Science. Prentice Hall, Englewood Cliffs (1998)
9. Oliveira, M.: *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, Department of Computer Science, University of York (2005)
10. Oliveira, M., Cavalcanti, A.: *ArcAngelC: a refinement tactic language for Circus*. *Electronic Notes in Theoretical Computer Science* 214, 203–229 (2008)
11. Oliveira, M., Cavalcanti, A., Woodcock, J.: *ArcAngel: a Tactic Language for Refinement*. *Formal Aspects of Computing* 15(1), 28–47 (2003)
12. Oliveira, M., Cavalcanti, A., Woodcock, J.: *A UTP semantics for Circus*. *Formal Aspects of Computing*, Online First (December 2007)
13. Oliveira, M., Cavalcanti, A., Zeyda, F.: *A Tactic Language for Refinement of State-Rich Concurrent Specifications* (to appear)
14. Oliveira, M., Xavier, M., Cavalcanti, A.: *Refine and Gabriel: Support for Refinement and Tactics*. In: *Proceedings of the Second Int. Conference on Software Engineering and Formal Methods*, pp. 310–319. IEEE Computer Society, Los Alamitos (2004)

15. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall Series in Computer Science. Prentice Hall, Englewood Cliffs (1997)
16. Spivey, J.M.: The Z Notation: A Reference Manual. Prentice-Hall International Series In Computer Science. Prentice Hall PTR, Englewood Cliffs (1992)
17. von Wright, J.: Program Refinement by Theorem Prover. In: BCS FACS Sixth Refinement Workshop – Theory and Practise of Formal Software Development, London, U.K. Springer, Heidelberg (1994)
18. Zeyda, F., Cavalcanti, A.: Supporting ArcAngel in ProofPower. Electronic Notes in Theoretical Computer Science 259, 225–243 (2009)
19. Zeyda, F., Cavalcanti, A.: Mechanical Reasoning about Families of UTP Theories. Science of Computer Programming (March 2010), doi:dx.doi.org/10.1016/j.scico.2010.02.010

Author Index

- Akhtar, Sabina 49
- Bauer, Sebastian S. 80
- Bendisposto, Jens 1
- Bidoit, Michel 80
- Bonsangue, Marcello 226
- Borba, Paulo 96
- Calegari, Daniel 112
- Caltais, Georgiana 226
- Cavalcanti, Ana 274
- Clarke, Edmund M. 144
- Crispim, Pedro 33
- da Silva, Paulo Salem 64
- de Melo, Ana C.V. 64
- Gawlitza, Thomas Martin 242
- Gervais, Frédéric 177
- Gheyi, Rohit 96
- Gomes, Artur O. 210
- Goriac, Eugen-Ioan 226
- Hennicker, Rolf 80
- Kleine, Moritz 128
- Laleau, Régine 177
- Le Métayer, Daniel 194
- Leuschel, Michael 1
- Liu, Zhiming 258
- Lopes, Antónia 33
- Lucanu, Dorel 226
- Luna, Carlos 112
- Madlener, Ken 161
- Massoni, Tiago 96
- Matoussi, Abderrahman 177
- Mazza, Eduardo 194
- Merz, Stephan 49
- Morisset, Charles 258
- Mostowski, Wojciech 17
- Oliveira, Marcel V.M. 210
- Poll, Erik 17
- Potet, Marie-Laure 194
- Quinson, Martin 49
- Rutten, Jan 226
- Sanders, J.W. 128
- Seidl, Helmut 242
- Silva, Alexandra 226
- Szasz, Nora 112
- Tamalet, Alejandro 161
- Tasistro, Álvaro 112
- Vasconcelos, Vasco T. 33
- Verma, Kumar Neeraj 242
- Wang, Shuling 258
- Younes, Håkan L.S. 144
- Zeyda, Frank 274
- Zuliani, Paolo 144